INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL’S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS’ FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document 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. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

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

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 2014 - 2020, Intel Corporation. All rights reserved.
## Contents

1. **Introduction** ......................................................................................................................... 9  
   1.1 **Purpose** .......................................................................................................................... 9  
   1.2 **Intended Audience** .......................................................................................................... 9  
   1.3 **Related Documents** ......................................................................................................... 9  

2. **FSP Overview** ......................................................................................................................... 10  
   2.1 **Design Philosophy** .......................................................................................................... 10  
   2.2 **Technical Overview** ......................................................................................................... 10  
   2.2.1 **Data Structure Descriptions** ...................................................................................... 10  

3. **FSP Integration** ....................................................................................................................... 11  
   3.1 **FSP Distribution Package** .............................................................................................. 11  

4. **FSP Binary Format** ................................................................................................................. 12  
   4.1.1 **FSP-T: Temporary RAM initialization phase** .............................................................. 12  
   4.1.2 **FSP-M: Memory initialization phase** .......................................................................... 12  
   4.1.3 **FSP-S: Silicon initialization phase** .............................................................................. 12  
   4.1.4 **OEM Components (FSP-O)** ....................................................................................... 13  
   4.2 **FSP Component Identification** ....................................................................................... 13  
   4.2.1 **FSP Image ID and Revision** ......................................................................................... 13  
   4.2.2 **FSP Component Layout** ............................................................................................ 13  

5. **FSP Information tables** .......................................................................................................... 15  
   5.1.1 **FSP_INFO_HEADER** .................................................................................................... 15  
   5.1.2 **FSP_INFO_EXTENDED_HEADER** .............................................................................. 17  
   5.1.3 **Locating FSP_INFO_HEADER** .................................................................................... 19  
   5.1.4 **FSP Description File** .................................................................................................. 21  
   5.1.5 **FSP Patch Table (FSPP)** ............................................................................................ 21  

6. **FSP Configuration Data** ........................................................................................................ 23  
   6.1 **UPD Standard Fields** ...................................................................................................... 24  
   6.1.1 **FSP-T UPD Structure** .................................................................................................. 24  
   6.1.2 **FSP-M UPD Structure** ................................................................................................ 26  
   6.1.3 **FSP-S UPD Structure** .................................................................................................. 27  

7. **Boot Flow** .............................................................................................................................. 29  
   7.1 **API Mode Boot Flow** ...................................................................................................... 29  
   7.1.1 **Boot Flow Description** ............................................................................................... 29  
   7.2 **Dispatch Mode Boot Flow** ............................................................................................... 31  
   7.2.1 **High Level Overview** ................................................................................................. 31  
   7.2.2 **Boot Flow Description** ............................................................................................... 32  
   7.2.3 **Alternate Boot Flow Description** ............................................................................... 34  

8. **FSP API Mode Interface** ........................................................................................................ 36  
   8.1 **Entry-Point Invocation Environment** ................................................................................ 36  
   8.2 **Data Structure Convention** ............................................................................................ 36  
   8.3 **Entry-Point Calling Convention** ...................................................................................... 36  
   8.4 **Return Status Code** ........................................................................................................ 36  
   8.5 **FSP Events** ...................................................................................................................... 37  
   8.5.1 **Related Definitions** .................................................................................................... 38
11 Other Host BootLoader Considerations ................................................. 66
11.1 ACPI ...................................................................................... 66
11.2 Bus Enumeration ..................................................................... 66
11.3 Security .................................................................................. 66
12 Appendix A – Data Structures ....................................................... 67
12.1 BOOT_MODE .......................................................................... 67
   12.1.1 PiBootMode.h .................................................................. 67
12.2 EFI_STATUS ............................................................................ 68
   12.2.1 UefiBaseType.h ................................................................ 68
   12.2.2 OEM Status Code .............................................................. 68
12.3 EFI_PEI_GRAPHICS_INFO_HOB .................................................. 70
   12.3.1 GraphicsInfoHob.h ............................................................. 70
12.4 EFI_PEI_GRAPHICS_DEVICE_INFO_HOB ...................................... 70
   12.4.1 GraphicsInfoHob.h ............................................................. 70
12.5 EFI_GUID ................................................................................ 70
   12.5.1 Base.h ........................................................................... 70
   12.5.2 UefiBaseType.h ................................................................. 71
12.6 EFI_MEMORY_TYPE ................................................................... 72
   12.6.1 UefiMultiPhase.h ............................................................... 72
12.7 Hand Off Block (HOB) ............................................................... 73
   12.7.1 PiHob.h ........................................................................... 73
12.8 Firmware Volume and Firmware Filesystem .............................. 76
   12.8.1 PiFirmwareVolume.h ......................................................... 76
   12.8.2 PiFirmwareFile.h ............................................................... 77
12.9 Debug Error Level .................................................................. 80
   12.9.1 DebugLib.h .................................................................... 80
12.10 Event Code Types ................................................................. 81
12.11 EFI_STATUS_CODE_STRING_DATA ......................................... 82
13 Appendix B – Acronyms ................................................................. 83

Figures

Figure 1: FSP Component Logical View ................................................. 12
Figure 2: FSP Component Layout View ............................................... 14
Figure 3: FSP Component Headers ..................................................... 20
Figure 4: API Mode Boot Flow ......................................................... 29
Figure 5: Dispatch Mode Boot Flow ................................................... 31
Figure 6: Dispatch Mode Design ....................................................... 53

Tables

Table 1. FSP_INFO_HEADER ................................................................ 15
Table 2. FSP_INFO_EXTENDED_HEADER ......................................... 17
Table 3. FSPP – PatchData Encoding .................................................. 21
Table 4. UPD Standard Fields ........................................................... 24
Table 5. Return Values - FspEventHandler() ...................................... 39
Table 6. Return Values - *TempRamInit()* API ................................................................. 42
Table 7. Return Values - *FspMemoryInit()* API ................................................................. 43
Table 8. Return Values - *TempRamExit()* API ................................................................. 45
Table 9. Return Values - *FspSiliconInit()* API ................................................................. 47
Table 10. Return Values - *FspMultiPhaseSilInit()* API ....................................................... 49
Table 11. Return Values - *NotifyPhase()* API ................................................................. 52
Table 12. Return Values - *TempRamExit()* PPI ................................................................. 57
## Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
</table>
| **May 2020** | 2.2      | - Based on FSP EAS v2.1 – Backward compatibility is retained.
|            |          | - Added multi-phase silicon initialization to increase the modularity of the *FspSiliconInit()* API. |
|            |          | - Added FSP event handlers. |
|            |          | - Added *FspMultiPhaseSilInit()* API |
|            |          | - **FSP_INFO_HEADER** changes |
|            |          |   - Updated *SpecVersion* from 0x21 to 0x22 |
|            |          |   - Updated *HeaderRevision* from 4 to 5 |
|            |          |   - Added *FspMultiPhaseSilInitEntryOffset* |
|            |          | - Added **FSPT_ARCH_UPD** |
|            |          |   - Added *FspDebugHandler* |
|            |          | - **FSPM_ARCH_UPD** changes |
|            |          |   - Added *FspEventHandler* |
|            |          | - **FSPS_ARCH_UPD** changes |
|            |          |   - Added *EnableMultiPhaseSiliconInit*, bootloaders designed for FSP 2.0/2.1 can disable the *FspMultiPhaseSilInit()* API and continue to use *FspSiliconInit()* without change. |
|            |          |   - Added *FspEventHandler* |

| **May 2019** | 2.1      | - Based on FSP EAS v2.0 – Backward compatibility is retained. |
|             |          | - Added Dispatch Mode to ease integration with UEFI bootloaders. |
|             |          | - **FSP_INFO_HEADER** changes |
|             |          |   - Updated *SpecVersion* from 0x20 to 0x21 |
|             |          |   - Updated *HeaderRevision* from 3 to 4 |
|             |          |   - Defined bit 1 in *ImageAttribute* to indicate support for dispatch mode. |
|             |          | - **FSPM_ARCH_UPD** changes |
|             |          |   - Modified *StackBase* and *StackSize* to only contain FSP heap data during pre-memory phase. |
|             |          | - FSP_GET_DEVICES() constants may now be returned by *NotifyPhase()* |
|             |          | - Added description of dispatch mode boot flow |
|             |          | - Added dispatch mode API definitions |
|             |          | - Added *FSP_ERROR_INFO* & *FSP_ERROR_INFO_HOB* |
|             |          | - Added *EFI_PEI_GRAPHICS_DEVICE_INFO_HOB* |

| **April 2016** | 2.0      | - 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 |
|               |          |   - Updated *HeaderRevision* from 2 to 3 |
|               |          |   - Reduced *ImageAttribute* field from 4 to 2 bytes |
- Defined new `ComponentAttribute` field and defined `ComponentType` (Bits15:12)
- Defined Bit0 and Bit1 in `ComponentAttribute` for Debug/Release & Test/Official respectively
- Renamed `Reserved` to `Reserved1`
- Renamed `ApiEntryNum` to `Reserved2`
- Renamed `FspInitEntryOffset` to `Reserved3`
- Added `SpecVersion` 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 `TempRamInit()`, `FspMemoryInit()`, `TempRamExit()`, `FspSiliconInit()` and `NotifyPhase()` API
  - `TempRamInit()`
    - Stack usage/stack allocation to bootloader clarified
    - Calling convention exception clarified
    - Removed parameter structure/description.
    - Updated API parameters to use FSPT_UPD
  - `FspMemoryInit()`
    - 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
  - `TempRamExit()` - Updated API parameters
  - `NotifyPhase()` - Added EndOfFirmware 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 firmware or UEFI firmware or BIOS developers, bootloader developers, system integrators, as well as end users.

1.3 Related Documents

- Intel® FSP EAS version 2.1
  https://cdrdv2.intel.com/v1/dl/getContent/611786
- Boot Specification File (BSF) Specification
- Unified Extensible Firmware Interface (UEFI) Specification
  http://www.uefi.org/specifications
- Platform Initialization (PI) Specification v1.7
  https://uefi.org/sites/default/files/resources/PI_Spec_1_7_final_Jan_2019.pdf
- Binary Configuration Tool (BCT) for Intel® Firmware Support Package - available at
  http://www.intel.com/fsp
- Intel® Firmware Module Management Tool (Intel® FMMT) – available at
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 operating systems.

Below are some required steps for the integration:

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

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

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

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

### 3.1 FSP Distribution Package

The FSP distribution package contains the following:

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

The Binary Configuration Tool (BCT) can be used to configure the FSP. BCT is available as a separate package.
4 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: Temporary RAM initialization phase

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

This phase consists of below FSP API

- `TempRamInit()`

4.1.2 FSP-M: Memory initialization phase

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

This phase consists of below FSP API

- `FspMemoryInit()`
- `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 API

- `FspSiliconInit()`
- `NotifyPhase()` -Post PCI bus enumeration, Ready To Boot and End of Firmware.
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. These components can be packaged in any order inside the FSP binary.
Figure 2: FSP Component Layout View

- **FSP Top**
  - FSP-T
  - Temp RAM Phase Data
  - FSP_INFO_HEADER - T
  - FSP-M
  - Memory Init Phase Data
  - FSP_INFO_HEADER - M
  - FSP-S
  - Silicon Init Phase Data
  - FSP_INFO_HEADER - S

- **FSP Base**
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 bytes 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>
<thead>
<tr>
<th>Table 1. FSP_INFO_HEADER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte Offset</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Byte Offset</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>36</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>44</td>
</tr>
<tr>
<td>48</td>
</tr>
</tbody>
</table>
### 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

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Size in Bytes</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Signature</td>
<td>‘FSPE’. Signature for the FSP_INFO_EXTENDED_HEADER.</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Length</td>
<td>Length of the table in bytes, including all additional FSP producer defined data.</td>
</tr>
</tbody>
</table>
### FSP Information Tables

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Size in Bytes</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>Revision</td>
<td>FSP producer defined revision of the table.</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>Reserved</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>FspProducerId</td>
<td>FSP producer identification string.</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>FspProducerRevision</td>
<td>FSP producer implementation revision number. Larger numbers are assumed to be newer revisions.</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>FspProducerDataSize</td>
<td>Size of the FSP producer defined data (n) in bytes.</td>
</tr>
<tr>
<td>24</td>
<td>n</td>
<td>...</td>
<td>FSP producer defined data of size (n) defined by FspProducerDataSize.</td>
</tr>
</tbody>
</table>
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.

```c
#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 need 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.

```c
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**

<table>
<thead>
<tr>
<th>BIT [23:00]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT [27:24]</td>
<td>Patch type</td>
</tr>
<tr>
<td></td>
<td>0000: Patch DWORD at OFFSET with the delta of the new and old base.</td>
</tr>
<tr>
<td></td>
<td>NewValue = OldValue + (NewBase - OldBase)</td>
</tr>
<tr>
<td></td>
<td>1111: Same as 0000</td>
</tr>
<tr>
<td></td>
<td>Others: Reserved</td>
</tr>
<tr>
<td>BIT [28:30]</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

May 2020
Firmware Support Package EAS
Document Number: 627153-2.2
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 0xFFF00000 to 0xFFF00000.

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

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

Looking closer at the patch table entries:

```
0xFFFFFFFFC, ///< 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 0xFFF00000, then the new value should be:

\[
\text{OldValue} + (\text{NewBase} - \text{OldBase}) = 0xFFF00000 + (0xFFF00000 - 0xFFF00000) = 0xFFF00000
\]

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 can be used by the FSP during initialization. This configuration region is a data structure called the Updateable Product Data (UPD) and will contain the default parameters for FSP initialization. The UPD data structure is only used by the FSP when the FSP is being invoked using the API mode interface defined in Section 8.

When the FSP is invoked according to the dispatch mode interface defined in Section 9, the UPD configuration region and the UPD data structure are not used by the FSP. In dispatch mode, the PPI database and PCD database are shared between the bootloader and the FSP. Because they are shared, the UPD configuration region is not needed to provide a mechanism to pass configuration data from the bootloader to the FSP. Instead, configuration data is communicated to the FSP using PCD and PPI. The bootloader may utilize the UPD to influence PCD and PPI contents provided to the FSP in dispatch mode.

The UPD parameters can be statically customized using a separate Binary 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 APIs', the FSP will use the default built-in UPD configuration data in the respective FSP components. However, if the bootloader needs to update any of the UPD parameters, it is recommended to copy the whole UPD structure from the FSP component to memory, update the parameters and initialize the UPD pointer to the address of the updated UPD structure. The FSP API will then use this data structure instead of the default configuration region data for platform initialization. The UPD data structure is a project specific structure. Please refer to the Integration Guide for the details of this structure.

The UPD structure has some standard fields followed by platform specific 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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 – 0x07</td>
<td>UPD Region Signature. The signature will be</td>
</tr>
<tr>
<td></td>
<td>“XXXXXX_T” for FSP-T</td>
</tr>
<tr>
<td></td>
<td>“XXXXXX_M” for FSP-M</td>
</tr>
<tr>
<td></td>
<td>“XXXXXX_S” for FSP-S</td>
</tr>
<tr>
<td></td>
<td>Where XXXXXX is a unique signature</td>
</tr>
<tr>
<td>0x08</td>
<td>Revision of the Data structure</td>
</tr>
<tr>
<td>0x09 – 0x1F</td>
<td>Reserved[23]</td>
</tr>
<tr>
<td>0x20 – n</td>
<td>Platform Specific Parameters, where the n is equal to</td>
</tr>
<tr>
<td></td>
<td>(FSP_INFO_HEADER.CfgRegionSize – 1)</td>
</tr>
</tbody>
</table>

```c
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;
    FSPT_ARCH_UPD FsptArchUpd;
}

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

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

<table>
<thead>
<tr>
<th>Revision</th>
<th>Revision of the structure is 1 for this version of the specification.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Length of the structure in bytes. The current value for this field is 32.</td>
</tr>
<tr>
<td>FspDebugHandler</td>
<td>Optional debug handler for the bootloader to receive debug messages occurring during FSP execution. Refer to Section 8.5 for more details.</td>
</tr>
</tbody>
</table>
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;
    FSP_EVENT_HANDLER FspEventHandler;
    UINT8 Reserved1[4];
} FSPM_ARCH_UPD;

<table>
<thead>
<tr>
<th><strong>Revision</strong></th>
<th>Revision of the structure is 2 for this version of the specification.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NvsBufferPtr</strong></td>
<td>Pointer to the non-volatile storage (NVS) data buffer. If it is <strong>NULL</strong> it indicates the NVS data is not available. Refer to Section 10.2 for more details.</td>
</tr>
<tr>
<td><strong>StackBase</strong></td>
<td>Pointer to the temporary RAM base address to be consumed inside <em>FspMemoryInit()</em> API. For FSP implementations compliant to v2.0 of this specification, the temporary RAM is used to establish a stack and a HOB heap. For FSP implementations compliant to v2.1 of this specification, the temporary RAM is only used for a HOB heap. Starting with v2.1 of this specification, FSP will run on top of the stack provided by the bootloader instead of establishing a separate stack. This allows the stack memory to be reused after <em>FspMemoryInit()</em> returns to the bootloader. To retain backwards compatibility with earlier versions of this specification, this parameter retains the <strong>StackBase</strong> name.</td>
</tr>
</tbody>
</table>
**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.

| **StackSize** | For FSP implementations compliant to v2.0 of this specification, the temporary RAM size used to establish a stack and HOB heap. Consumed by the FspMemoryInit() API. For FSP implementations compliant to v2.1 of this specification, the temporary RAM size used to establish a HOB heap inside the FspMemoryInit() API. Starting with v2.1 of this specification, FSP will run on top of the stack provided by the bootloader instead of establishing a separate stack. This allows the stack memory to be reused after FspMemoryInit() returns to the bootloader. To retain backwards compatibility with earlier versions of this specification, this parameter retains the **StackSize** name. Refer to the Integration Guide for the minimum required temporary RAM size. In the case of FSP v2.1, the Integration Guide shall also specify the minimum free stack space required at the point where the FSP API entrypoints are called. |
| **BootloaderTolumSize** | Size of memory to be reserved by FSP below "top of low usable memory" for bootloader usage. Refer to Section 10.3 for more details. |
| **BootMode** | Current boot mode. Values are defined in Section 12.1 Appendix A – Data Structures. Refer to the Integration Guide for supported boot modes. |
| **FspEventHandler** | Optional event handler for the bootloader to be informed of events occurring during FSP execution. Refer to Section 8.5 for more details. This value is only valid if Revision is >= 2. |
typedef struct {
    FSP_UPD_HEADER        UpdHeader;
    FSPS_ARCH_UPD         FspsArchUpd;
}
FSPS_UPD;

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

<table>
<thead>
<tr>
<th><strong>Revision</strong></th>
<th>Revision of the structure is 1 for this version of the specification.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Length</strong></td>
<td>Length of the structure in bytes. The current value for this field is 32.</td>
</tr>
<tr>
<td><strong>FspEventHandler</strong></td>
<td>Optional event handler for the bootloader to be informed of events occurring during FSP execution. Refer to Section 8.5 for more details.</td>
</tr>
<tr>
<td><strong>EnableMultiPhaseSiliconInit</strong></td>
<td>An FSP binary may optionally implement multi-phase silicon initialization, see Section 8.10 for further details. This is only supported if the FspMultiPhaseSiInitEntryOffset field in <strong>FSP_INFO_HEADER</strong> is non-zero, see Section 5.1.1 for further details. To enable multi-phase silicon initialization, the bootloader must set <strong>EnableMultiPhaseSiliconInit</strong> to a non-zero value.</td>
</tr>
</tbody>
</table>
7 Boot Flow

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

7.1 API Mode Boot Flow

Figure 4: API Mode Boot Flow

7.1.1 Boot Flow Description

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 the `TempRamInit()` API. The bootloader also has the option to initialize the temporary memory directly, in which case this step and step 2 are skipped.

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 in step 1.c)... this step and the next step are skipped.

7. FSP returns from `TempRamExit()` API.

8. Bootloader finds FSP-S and calls `FspSiliconInit()` API.

9. FSP returns from `FspSiliconInit()` API.

10. If supported by the FSP and the bootloader enables multi-phase silicon initialization by setting `FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit` to a non-zero value:

    a) Bootloader calls the `FspMultiPhaseSiInit()` API with the `EnumMultiPhaseGetNumberOfPhases` parameter to discover the number of silicon initialization phases supported by the bootloader.

    b) Bootloader must call the `FspMultiPhaseSiInit()` API with the `EnumMultiPhaseExecutePhase` parameter n times, where n is the number of phases returned previously. Bootloader may perform board specific code in between each phase as needed.

    c) The number of phases, what is done during each phase, and anything the bootloader may need to do in between phases shall be described in the Integration Guide.

11. Bootloader continues and device enumeration.


13. Bootloader calls `NotifyPhase()` API with `ReadyToBoot` parameter before transferring control to OS loader.

14. When booting to a non-UEFI OS, Bootloader calls `NotifyPhase()` API with `EndOfFirmware` parameter immediately after `ReadyToBoot`. 
15. When booting to a UEFI OS, Bootloader calls NotifyPhase() with EndOfFirmware parameter during ExitBootServices.

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

### 7.2 Dispatch Mode Boot Flow

Dispatch mode is an optional boot flow intended to enable FSP to integrate well in to UEFI bootloader implementations. Implementation of this boot flow necessitates that the underlying FSP implementation uses the Pre-EFI Initialization (PEI) environment defined in the *PI Specification*. It is possible to implement an FSP without using PEI, so bootloaders must check that dispatch mode is available using the `FSP_INFO_HEADER`, see Section 5.1.1 for further details. The *Integration Guide* will also specify if an FSP implements dispatch mode. See Section 9 for a full description of dispatch mode.

#### 7.2.1 High Level Overview

**Figure 5: Dispatch Mode Boot Flow**

![Diagram](image)

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

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

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

7.2.2 Boot Flow Description

This boot flow assumes that the bootloader is a typical UEFI firmware implementation conforming to the PI Specification. Therefore, the bootloader will follow the standard four phase PI boot flow progressing from SEC phase, to PEI phase, to DXE phase, to BDS phase.

1. Bootloader provided SEC phase 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 the TempRamInit() API. SEC also has the option to initialize the temporary memory directly, in which case this step and step 2 are skipped.

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

3. SEC initializes the stack in temporary memory.

4. SEC finds FSP-M and adds an instance of EFI_PEI_CORE_FV_LOCATION_PPI containing the address of FSP-M to the PpiList passed in to PEI core.

5. SEC calls the entry point for the PEI core inside FSP-M.
   a) Boot loader passes the FSP-M PEI core a EFI_SEC_PEI_HAND_OFF data structure with the BootFirmwareVolumeBase and BootFirmwareVolumeSize members pointing to a FV provided by the platform.
      ▪ The bootloader provides the Boot Firmware Volume (BFV). Consequently, in FSP dispatch mode PEI core is not in the BFV unlike most UEFI firmware implementations.

6. PEI core dispatches the PEIM in the BFV provided by the bootloader.

7. Bootloader installs FSPM_ARCH_CONFIG_PPI.

8. One of the PEIM provided by the bootloader installs a EFI_PEI_FIRMWARE_VOLUME_INFO_PPI for each FV contained in FSP-M.
   a) The bootloader must not install the EFI_PEI_FIRMWARE_VOLUME_INFO_PPI(s) for FSP-M until the bootloader is ready for FSP-M to execute.
b) If FSP-M requires any DynamicEx PCD values, the bootloader must ensure those PCD contain valid data before installing the `EFI_PEI_FIRMWARE_VOLUME_INFO_PPI` for FSP-M.

9. PEI core will continue to dispatch PEIM. During the course of dispatch, PEIM included with FSP-M will be executed.

   a) Some of the PEIM contained in FSP-M may require configuration data to be provided by the bootloader. If this is the case, the configuration data may be stored in either DynamicEx PCD or PPI.

      ▪ If the configuration data is stored in PCD, then it is assumed that the PCD contains valid data before FSP-M begins execution.

      ▪ If the configuration data is stored in PPI, then the needed PPI will either be in the PEIM’s DEPEX, or the PEIM will register a callback for the needed PPI and not attempt to access the PPI until the callback is invoked by PEI core.

10. FSP-M installs `FSP_TEMP_RAM_EXIT_PPI`.

11. After dispatching the PEIM in FSP-M, memory will be initialized. Accordingly, FSP-M will call `(*PeiServices)->InstallPeiMemory()`.

   a) PEI core shadows to main memory.

   b) PEI core invokes `TemporaryRamDone()` from `EFI_PEI_TEMPORARY_RAM_DONE_PPI`. The implementation of `EFI_PEI_TEMPORARY_RAM_DONE_PPI` is provided by the bootloader.

   c) The bootloader implementation of `EFI_PEI_TEMPORARY_RAM_DONE_PPI` calls `TempRamExit()` from `FSP_TEMP_RAM_EXIT_PPI`.

      ▪ For platforms that use the SEC implementation in UefiCpuPkg, SEC core implements `EFI_PEI_TEMPORARY_RAM_DONE_PPI`. The `TemporaryRamDone()` implementation in SEC core will call `SecPlatformDisableTemporaryMemory()`. This function would then locate `FSP_TEMP_RAM_EXIT_PPI` and call `TempRamExit()`.

      ▪ If the bootloader did not call `TempRamInit()` in step 1.c) then the bootloader would not call `TempRamExit()`.

   d) PEI core follows up with an installation of the `EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI`. Refer to Volume 1 of the PI Specification for details.

12. Post memory PEIM provided by the bootloader are now executed.

13. One of the PEIM provided by the bootloader installs a `EFI_PEI_FIRMWARE_VOLUME_INFO_PPI` for each FV contained in FSP-S.
Boot Flow

a) The bootloader must not install the `EFI_PEI_FIRMWARE_VOLUME_INFO_PPI`(s) for FSP-S until the bootloader is ready for FSP-S to execute.

b) If FSP-S requires any DynamicEx PCD values, the bootloader must ensure those PCD contain valid data before installing the `EFI_PEI_FIRMWARE_VOLUME_INFO_PPI`(s) for FSP-S.

14. PEI core will continue to dispatch PEIM. During the course of dispatch, PEIM included with FSP-S will be executed.

a) Some of the PEIM contained in FSP-S may require configuration data to be provided by the bootloader. If this is the case, the configuration data may be stored in either DynamicEx PCD or PPI.

- If the configuration data is stored in PCD, then it is assumed that the PCD contain valid data before FSP-S begins execution.

- If the configuration data is stored in PPI, then the needed PPI will either be in the PEIM’s DEPEX, or the PEIM will register a callback for the needed PPI and not attempt to access the PPI until the callback is invoked by PEI core.

15. End of PEI is reached, and DXE begins execution.

16. Any DXE drivers included in FSP-S are dispatched. These drivers may create events to be notified at different points in the boot flow. FSP shall use a subset of the events defined by the PI Specification, see Section 9.3 for the full list of events the FSP may use.

17. DXE signals `EFI_END_OF_DXE_EVENT_GROUP_GUID` and transitions to BDS phase.

a) Note: The PI Specification does not require that Step 17 occurs before Step 18, however most implementations appear to use this order.

18. BDS starts the PCI bus driver, which enumerates PCI devices. After enumeration, the PCI bus driver installs the `EFI_PCI_ENUMERATION_PROTOCOL`. DXE signals any applicable events.

19. BDS signals `EFI_EVENT_GROUP_READY_TO_BOOT` immediately before loading the OS boot loader.

20. BDS executes the OS boot loader. The OS boot loader loads the OS kernel into memory.

21. The OS boot loader calls `ExitBootServices()`, DXE signals this event before shutting down the UEFI Boot Services.

7.2.3 Alternate Boot Flow Description

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

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

1. Bootloader provided SEC phase 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 the \texttt{TempRamInit()} API. SEC also has the option 
      to initialize the temporary memory directly, in which case this step 
      and step 2 are skipped.

2. FSP initializes temporary memory and returns from \texttt{TempRamInit()} API.

3. SEC initializes the stack in temporary memory.

4. SEC calls the entry point for the PEI core inside the \textit{Boot Firmware Volume} 
   (BFV).

5. PEI core dispatches the PEIM in the BFV provided by the bootloader.

6. Boot loader installs \texttt{FSPM\_ARCH\_CONFIG\_PPI}.

7. One of the PEIM provided by the bootloader installs a 
   \texttt{EFI\_PEI\_FIRMWARE\_VOLUME\_INFO\_PPI} for each FV contained in FSP-M.
   
   a) The bootloader must not install the 
      \texttt{EFI\_PEI\_FIRMWARE\_VOLUME\_INFO\_PPI}(s) for FSP-M until the 
      bootloader is ready for FSP-M to execute.
   
   b) If FSP-M requires any DynamicEx PCD values, the bootloader must 
      ensure those PCD contain valid data before installing the 
      \texttt{EFI\_PEI\_FIRMWARE\_VOLUME\_INFO\_PPI}(s) for FSP-M.

8. PEI core will encounter a second PEI core in FSP-M. Because it is not a PEIM, 
   the dispatcher will skip it. PEI core will proceed to dispatch the PEIM in FSP-M.

9. The boot flow proceeds the same as step 9 in the primary boot flow from here 
   forwards.
8 FSP API Mode Interface

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.
- Sufficient stack space should be available for the FSP API function to execute. Consult the Integration Guide for platform specific stack space requirements.

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 API 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 12.2.1 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 12.2.2 OEM Status code.

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

### 8.5 FSP Events

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

The bootloader may provide an event handler to the FSP through the **FSPM_ARCH_UPD.FspEventHandler** and **FSPS_ARCH_UPD.FspEventHandler** UPDs. Providing these event handlers is entirely optional. If the bootloader does not wish to handle FSP events, it may set these UPDs to **NULL**. FSP will only call **FSPM_ARCH_UPD.FspEventHandler** during FSP-M and **FSPS_ARCH_UPD.FspEventHandler** during FSP-S.

The FSP may use this event mechanism to provide debug log messages to the bootloader. When FSP-M or FSP-S provide debug log messages this way, the **Type** parameter’s **EFI_STATUS_CODE_TYPE_MASK** will be set to **EFI_DEBUG_CODE** and the **Data** parameter shall contain a **EFI_STATUS_CODE_STRING_DATA** payload. Please see section 6.6.2 of the *PI Specification v1.7 Volume 3* for details on **EFI_STATUS_CODE_STRING_DATA**. The FSP shall only pass a **EFI_STRING_TYPE** of **EfiStringAscii** for the purposes of debug log messages. The Instance parameter shall contain the **ErrorLevel**, please see section 12.9 for details. The bootloader may parse these debug log events if desired.

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

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

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

Due to the nature of early boot stages, FSP-T is mostly assembly code. Accordingly, FSP-T uses a more simple interface that only provides debug log messages using **FSPT_ARCH_UPD.FspDebugHandler**. Due to the need for a stack to be established to call this handler, FSP-T can only call **FspDebugHandler()** after temporary memory is initialized. This may delay the output of debug log messages until later in the FSP-T flow.
The event handlers provided by the bootloader should not use more than 4KB of stack space.

A similar feature is provided for dispatch mode, see Section 9.4.7.

8.5.1 Related Definitions
#define FSP_EVENT_CODE 0xF5000000
#define FSP_POST_CODE (FSP_EVENT_CODE | 0x00F80000)

See Section 12.10-12.11 Appendix A – Data Structures for the definitions of EFI_STATUS_CODE_TYPE, EFI_STATUS_CODE_VALUE, and EFI_STATUS_CODE_DATA.

8.5.1.1 FspEventHandler
Handler for FSP events, provided by the bootloader.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>Indicates the type of event being reported. See Section 12.10 Appendix A – Data Structures for the definition of <code>EFI_STATUS_CODE_TYPE</code>.</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>Describes the current status of a hardware or software entity. This includes information about the class and subclass that is used to classify the entity as well as an operation. For progress events, the operation is the current activity. For error events, it is the exception. For debug events, it is not defined at this time. See Section 12.10 Appendix A – Data Structures for the definition of <code>EFI_STATUS_CODE_VALUE</code>.</td>
</tr>
<tr>
<td><strong>Instance</strong></td>
<td>The enumeration of a hardware or software entity within the system. A system may contain multiple entities that match a class/subclass pairing. The instance differentiates between them. An instance of 0 indicates that instance information is unavailable, not meaningful, or not relevant. Valid instance numbers start with 1.</td>
</tr>
<tr>
<td><strong>CallerId</strong></td>
<td>This parameter can be used to identify the sub-module within the FSP generating the event. This parameter may be <code>NULL</code>.</td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td>This optional parameter may be used to pass additional data. The contents can have event-specific data. For example, the FSP provides a <code>EFI_STATUS_CODE_STRING_DATA</code> instance to this parameter when sending debug messages. This parameter is <code>NULL</code> when no additional data is provided. See Section 12.11 Appendix A – Data Structures for the definition of <code>EFI_STATUS_CODE_STRING_DATA</code>.</td>
</tr>
</tbody>
</table>

8.5.1.1.3 Return Values

The return status will be passed back through the `EAX` register.

**Table 5. Return Values - FspEventHandler()**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>The event was handled successfully.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td>Input parameters are invalid.</td>
</tr>
<tr>
<td><code>EFI_DEVICE_ERROR</code></td>
<td>The event handler failed.</td>
</tr>
</tbody>
</table>
8.5.1.2 **FspDebugHandler**

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

8.5.1.2.1 **Prototype**

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

8.5.1.2.2 **Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DebugMessage</td>
<td>A pointer to the debug message to be written to the log.</td>
</tr>
<tr>
<td>MessageLength</td>
<td>Number of bytes to written to the debug log.</td>
</tr>
</tbody>
</table>

8.5.1.2.3 **Return Values**

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

8.6.1 Prototype

typedef
EFI_STATUS
(EIFIAPI *FSP_TEMP_RAM_INIT) ( 
    IN VOID *FsptUpdDataPtr
);

8.6.2 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. |
8.6.3 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 6. Return Values - TempRamInit() API

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Temporary RAM was initialized successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Input parameters are invalid.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The FSP calling conditions were not met.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Temp RAM initialization failed.</td>
</tr>
</tbody>
</table>

8.6.4 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.
8.7 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.

8.7.1 Prototype

typedef

EFI_STATUS

(EIFIAPI *FSP_MEMORY_INIT) (  
    IN VOID   *FspmUpdDataPtr
    OUT VOID  **HobListPtr;
);

8.7.2 Parameters

<table>
<thead>
<tr>
<th>FspmUpdDataPtr</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HobListPtr</td>
<td>Pointer to receive the address of the HOB list as defined in the Section 12.7 - Appendix A – Data Structures</td>
</tr>
</tbody>
</table>

8.7.3 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 7. Return Values - FspMemoryInit() API

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>FSP execution environment was initialized successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Input parameters are invalid.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The FSP calling conditions were not met.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>FSP memory initialization failed.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Stack range requested by FSP is not met.</td>
</tr>
<tr>
<td>FSP_STATUS_RESET_REQUIRED_*</td>
<td>A reset is required. These status codes will not be returned during S3. See section 12.2.2 for details.</td>
</tr>
</tbody>
</table>
8.7.4 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 10.2 `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` statuses as described in section 8.4. This API will not request a reset during S3 resume flow.
8.8 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.

8.8.1 Prototype

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

8.8.2 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. |

8.8.3 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

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>FSP execution environment was initialized successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Input parameters are invalid.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The FSP calling conditions were not met.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Temporary memory exit.</td>
</tr>
</tbody>
</table>
8.8.4  **Description**

This API should be called only once after the `FspMemoryInit()` API and before `FspSiliconInit()` API.

This API tears down the temporary memory area set up in the cache and returns the cache to normal mode of operation. After the cache is returned to normal mode of operation, any data that was in the temporary memory is destroyed. It is therefore expected that the bootloader migrates any bootloader specific data that it might 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.

8.9  **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.

8.9.1  **Prototype**

```c
typedef EFI_STATUS (EFI_API *FSP_SILICON_INIT) (IN VOID *FspUpdDataPtr);
```

8.9.2  **Parameters**

| FspUpdDataPtr | Pointer to the FSPS_UPD data structure. If NULL, FSP will use the default parameters. Refer to the Integration Guide for structure definition. |
8.9.3 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

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>FSP execution environment was initialized successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Input parameters are invalid.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The FSP calling conditions were not met.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>FSP silicon initialization failed.</td>
</tr>
<tr>
<td>FSP_STATUS_RESET_REQUIRED_*</td>
<td>A reset is required. These status codes will not be</td>
</tr>
<tr>
<td></td>
<td>returned during S3.</td>
</tr>
</tbody>
</table>

8.9.4 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.

8.10 FspMultiPhaseSiInit API

This FSP API provides multi-phase silicon initialization; which brings greater modularity beyond the existing FspSiliconInit() API. Increased modularity is achieved by adding an extra API to FSP-S. This allows the bootloader to add board specific initialization steps throughout the SiliconInit flow as needed.

When using multi-phase silicon initialization, the FspSiliconInit() API is always called first; it is the first phase of silicon initialization. After the first phase, subsequent phases are invoked by calling the FspMultiPhaseSiInit() API.

This API may only be called after the FspSiliconInit() API and before NotifyPhase() API, and may not be called at any other time. This FSP API is optional and may not be implemented by all FSPs. Additionally, bootloaders may choose to not use it.
8.10.1 **Prototype**

typedef

EFI_STATUS

(EIFIAPIC *FSP_MULTI_PHASE_SI_INIT) (  
    IN FSP_MULTI_PHASE_PARAMS  MultiPhaseSiInitParamPtr
);

8.10.2 **Parameters**

| MultiPhaseSiInitParamPtr | Pointer to the FSP_MULTI_PHASE_PARAMS data structure. |

8.10.3 **Related Definitions**

typedef enum {
    EnumMultiPhaseGetNumberOfPhases = 0x0,
    EnumMultiPhaseExecutePhase = 0x1
} FSP_MULTI_PHASE_ACTION;

typedef struct {
    UINT32 NumberOfPhases;
    UINT32 PhasesExecuted;
} FSP_MULTI_PHASE_GET_NUMBER_OF_PHASES_PARAMS;

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

**EnumMultiPhaseGetNumberOfPhases**

This action returns the number of SiliconInit phases that the FSP supports. This indicates the maximum number of times the FspMultiPhaseSiInit() API may be called by the bootloader with the EnumMultiPhaseExecutePhase action given.

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

The EnumMultiPhaseGetNumberOfPhases action can be invoked by the bootloader as
many times as desired at any point between FspSiliconInit() and NotifyPhase(). It only
retrieves the current status, it does not modify it.

**EnumMultiPhaseExecutePhase**

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

### 8.10.4 Return Values

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

<table>
<thead>
<tr>
<th>EFI_SUCCESS</th>
<th>FSP execution environment was initialized successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Input parameters are invalid.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The FSP calling conditions were not met.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>FSP silicon initialization failed.</td>
</tr>
<tr>
<td>FSP_STATUS_RESET_REQUIRED_*</td>
<td>A reset is required. These status codes will not be returned during S3.</td>
</tr>
</tbody>
</table>

### 8.10.5 Description

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

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

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

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

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

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

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

If an initialization step requires a reset to take effect, the FspMultiPhaseSiInit() API will return an FSP_STATUS_RESET_REQUIRED as described in section 8.4. This API will not request a reset during S3 resume flow.

8.11 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

8.11.1 Prototype

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

8.11.2 Parameters

| NotifyPhaseParamPtr | Address pointer to the NOTIFY_PHASE_PARAMS |
8.11.3 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.
8.11.4 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>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The notification was handled successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The notification was not called in the proper order.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The notification code is invalid.</td>
</tr>
<tr>
<td>FSP_STATUS_RESET_REQUIRED_*</td>
<td>A reset is required. These status codes will not be returned during S3.</td>
</tr>
</tbody>
</table>

8.11.5 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.

If an initialization step requires a reset to take effect, the NotifyPhase() 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.
9  FSP Dispatch Mode Interface

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

### 9.1 Dispatch Mode Design

![Diagram of Dispatch Mode Design](image)

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

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

UPDs are not needed to provide a mechanism to pass configuration data from the bootloader to the FSP. Instead, configuration data is communicated to the FSP using PCDs and PPIs. These mechanisms are native to bootloader implementations conforming to the **PI Specification** and constitute a more natural method of supplying configuration data to the FSP. These PCDs and PPIs are platform specific. The **FSP Distribution Package** will contain source code definitions of the configuration data structures consumed by the FSP. The configuration data structures will also be described by the **Integration Guide**.
The bootloader must provide the PCD database implementation. Any dynamic PCDs consumed by the FSP must be included in the PCD database provided by the bootloader. The FSP Distribution Package will contain a DSC file which defines all PCDs used by the FSP. The recommended method of including these PCDs is to use the `!include` directive in the bootloader’s top-level platform DSC file. Because the FSP is a pre-compiled binary, all dynamic PCDs consumed by the FSP must be of the DynamicEx type. Refer to MdeModulePkg/Universal/PCD/Pei/Pcd.inf for more details on platform token numbers. In addition to the DSC file included in the FSP Distribution Package, the Integration Guide will also list the PCDs (along with TokenSpace GUID and TokenNumber) consumed by the FSP.

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

### 9.2 PEI Phase Requirements

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

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

- EFI_PEI_READ_ONLY_VARIABLE2_PPI

### 9.3 DXE and BDS Phase Requirements

DXE drivers contained in FSP firmware volumes are intended to be executed within the processor context and calling conventions defined by the PI Specification, Volume 2 for x64 platforms.

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

- ExitBootServices()
- SetWatchdogTimer()
- GetVariable()
- GetNextVariableName()
- SetVariable()
- QueryVariableInfo()
- setTime()
- SetWakeupTime()
- UpdateCapsule()
- QueryCapsuleCapabilities()
In addition, FSP may use the following *PI Specification* defined events during DXE phase:

1. **EFI_END_OF_DXE_EVENT_GROUP_GUID** – The *PI Specification* requires the bootloader to signal this event prior to invoking any UEFI drivers or applications that are not from the platform manufacturer, or connecting consoles.

2. **EFI_PCI_ENUMERATION_PROTOCOL** – The *PI Specification* requires the bootloader to install this protocol after PCI enumeration is complete.

3. **EFI_EVENT_GROUP_READY_TO_BOOT** – The *PI Specification* requires the bootloader to signal this event when it is about to load and execute a boot option.

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

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

### 9.4 Dispatch Mode API

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

#### 9.4.1 TempRamInit API

The *PI Specification* defines a code module format for PEI and DXE (PEIMs, and DXE Drivers respectively.) However, the *PI Specification* does not define a module format for SEC phase. Temporary RAM must be initialized during the SEC phase. Therefore, in dispatch mode FSP-T uses the same API defined in Section 8.6 to provide `TempRamInit()` to the bootloader SEC implementation.

#### 9.4.2 EFI PEI Core Firmware Volume Location PPI

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

#### 9.4.3 FSP Temporary RAM Exit PPI

`FSP_TEMP_RAM_EXIT_PPI`


9.4.3.1 Summary

Tears down the temporary memory set up by `TempRamInit` API.

9.4.3.2 GUID

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

9.4.3.3 Prototype

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

9.4.3.4 Parameters

<table>
<thead>
<tr>
<th><strong>TempRamExit</strong></th>
<th>Tears down the temporary memory set up by <code>TempRamInit</code> API.</th>
</tr>
</thead>
</table>

9.4.3.5 Description

This PPI provides the equivalent functionality as the `TempRamExit` function defined in Section 8.8 to bootloaders that use the FSP in dispatch mode. The `TempRamExit` function defined in this PPI tears down the temporary memory set up by `TempRamInit` API. Bootloaders that use dispatch mode must not use the `TempRamExit` API defined in Section 8.8, they must use this PPI instead.

9.4.4 FSP_TEMP_RAM_EXIT_PPI.TempRamExit()

9.4.4.1 Summary

Tears down the temporary memory set up by `TempRamInit` API.
Prototype

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

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. |

Description

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

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

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

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

Return Values

Table 12. Return Values - TempRamExit() PPI

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>FSP execution environment was initialized successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Input parameters are invalid.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The FSP calling conditions were not met.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Temporary memory exit.</td>
</tr>
</tbody>
</table>
9.4.5 **FSP-M Architectural Configuration PPI**

FSPM_ARCH_CONFIG_PPI

9.4.5.1 **Summary**

Architectural configuration data for FSP-M.

9.4.5.2 **GUID**

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

9.4.5.3 **Prototype**

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

9.4.5.4 **Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Revision</strong></td>
<td>Revision of the structure is 1 for this version of the specification.</td>
</tr>
<tr>
<td><strong>NvsBufferPtr</strong></td>
<td>Pointer to the non-volatile storage (NVS) data buffer. If it is NULL it indicates the NVS data is not available. Refer to Section 10.2 for more details.</td>
</tr>
<tr>
<td><strong>BootloaderTolumSize</strong></td>
<td>Size of memory to be reserved by FSP below &quot;top of low usable memory&quot; for bootloader usage. Refer to Section 10.3 for more details.</td>
</tr>
</tbody>
</table>

9.4.5.5 **Description**

This PPI contains architectural configuration data that is needed by PEIMs in FSP-M and/or FSP-S. It is the responsibility of the bootloader to install this PPI. The bootloader must be able to provide these data within the pre-memory PEI timeframe. In adherence with the weak ordering requirement for PEIMs, any PEIM contained in FSP that uses this PPI shall either include this PPI in its DEPEX or shall register a callback using (*PeiServices)-&gt;NotifyPpi () and refrain from accessing these data until the callback is invoked by the PEI Foundation.
As a performance optimization, it is recommended (but not required) that the boot
loader install this PPI before installing `EFI_PEI_FIRMWARE_VOLUME_INFO_PPI`
instances for the firmware volume(s) contained in FSP-M. This will reduce the number
of times the PEI Dispatcher will need to loop in order to complete PEI phase.

9.4.6 FSP Error Information

FSP_ERROR_INFO

9.4.6.1 Summary

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

9.4.6.2 GUID

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

9.4.6.3 Prototype

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

9.4.6.4 Parameters

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataHeader</td>
<td>The data header identifying the data. <code>DataHeader.HeaderSize</code> shall be <code>sizeof (EFI_STATUS_CODE_DATA)</code>. <code>DataHeader.Size</code> shall be <code>sizeof (FSP_ERROR_INFO) - HeaderSize</code>. Finally, <code>DataHeader.Type</code> shall be <code>STATUS_CODE_DATA_TYPE_FSP_ERROR_GUID</code>.</td>
</tr>
<tr>
<td>ErrorType</td>
<td>A GUID identifying the nature of the fatal error. This GUID is platform specific. A listing of all possible GUIDs shall be provided by the <code>Integration Guide</code>.</td>
</tr>
<tr>
<td>Status</td>
<td>A code describing the error encountered. Please see section 12.2 for a listing of possible error codes.</td>
</tr>
</tbody>
</table>

9.4.6.5 Description

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

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

9.4.7 FSP Debug Messages

FSP may optionally include the capability of generating log messages to aid in the debugging of firmware issues. When technically feasible, these log messages will be broadcast to the bootloader from the FSP by calling (*PeiServices)->ReportStatusCode() in PEI phase or EFI_STATUS_CODE_PROTOCOL.ReportStatusCode() in DXE phase. ReportStatusCode() will be called with the Type parameter's EFI_STATUS_CODE_TYPE_MASK set to EFI_DEBUG_CODE and the Data parameter containing a EFI_STATUS_CODE_STRING_DATA payload. Please see section 6.6.2 of the PI Specification v1.7 Volume 3 for details on EFI_STATUS_CODE_STRING_DATA. The FSP shall only pass a EFI_STRING_TYPE of EfiStringAscii for the purposes of debug log messages. The Instance parameter shall contain the ErrorLevel, please see section 12.9 for details. The bootloader may register a listener for these status codes if debug log messages are of interest.

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

Early in PEI, ReportStatusCode() may not be initialized. During this time, FSP may provide debug log messages using FSPT_ARCH_UPD.FspDebugHandler.

The FSP may also use ReportStatusCode() to provide POST codes to the bootloader. If FSP provides POST codes this way, the Type parameter's EFI_STATUS_CODE_TYPE_MASK will be set to EFI_PROGRESS_CODE and the Value parameter will have the upper 16-bits (EFI_STATUS_CODE_CLASS_MASK and EFI_STATUS_CODE_SUBCLASS_MASK) will be set to FSP_POST_CODE, see Section 8.5.1. The lower 16-bits (EFI_STATUS_CODE_OPERATION_MASK) will contain the POST code.
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 FSP-M.

In addition to the *PI Specification* defined HOB, the FSP produces a number of FSP architecturally defined GUID types 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*. 
10.1 FSP_RESERVED_MEMORY_RESOURCE_HOB

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

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

```c
#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.

10.2 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:

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

The bootloader needs to parse the HOB list to see if such a GUID HOB exists after memory is initialized. The HOB shall be populated either after returning from FspMemoryInit() in API mode or after all notification call backs for **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI** to be completed in dispatch mode. If it exists, the bootloader should extract the data portion from the HOB structure and then save it into a platform-specific NVS device, such as flash, EEPROM, etc. On the 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 FspMemoryInit() in API mode or FSPM_ARCH_CONFIG_PPI.NvsBufferPtr before installing FSPM_ARCH_CONFIG_PPI in dispatch mode. If the NVS device is memory mapped, the bootloader can initialize the buffer pointer directly to the buffer.

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

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.
10.3 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` when in API mode or `FSPM_ARCH_CONFIG_PPI.BootLoaderTolumSize` when in dispatch mode. The FSP reserved memory region will be placed below this region.

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

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

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

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

10.4 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.

```c
#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.

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

10.5 EFI_PEI_GRAPHICS_DEVICE_INFO_HOB

If BIT0 (Graphics Support) of the ImageAttribute field in the FSP_INFO_HEADER is set, the FSP includes graphics initialization capabilities. To complete the initialization of 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_DEVICE_INFO_HOB as described in the PI Specification as referenced in Section 1.3 Related Documents, which provides information about the graphics hardware which produces the framebuffer supplied by EFI_PEI_GRAPHICS_INFO_HOB.

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

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

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

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

10.6 FSP_ERROR_INFO_HOB

In the case of an error occurring during the execution of the FSP, the FSP may optionally produce an FSP_ERROR_INFO_HOB which describes the error in more detail. This HOB is only produced in API mode. In dispatch mode, ReportStatusCode () is used as described in section 9.4.6.

#define FSP_ERROR_INFO_HOB_GUID \ {0x611e6a88, 0xad7b, 0x4301, \ {0x93, 0xff, 0xe4, 0x73, 0x04, 0xb4, 0x3d, 0xa6}}

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

<table>
<thead>
<tr>
<th>GuidHob</th>
<th>The GUID HOB header identifying the data. GuidHob.Name shall be FSP_ERROR_INFO_HOB_GUID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>A ReportStatusCode() type identifier. The Type’s EFI_STATUS_CODE_TYPE_MASK must be EFI_ERROR_CODE with the EFI_STATUS_CODE_SEVERITY_MASK &gt;= EFI_ERROR_UNRECOVERED. See Section 6 of the PI Specification v1.7 Volume 3.</td>
</tr>
<tr>
<td>Value</td>
<td>A ReportStatusCode() Value. Used to determine status code class and sub-class, see Section 6 of the PI Specification v1.7 Volume 3. This field shall be set to zero (0).</td>
</tr>
<tr>
<td>Instance</td>
<td>A ReportStatusCode() Instance number. See Section 6 of the PI Specification v1.7 Volume 3. This field shall be set to zero (0).</td>
</tr>
<tr>
<td>CallerId</td>
<td>An optional GUID which may be used to identify which internal component of the FSP was executing at the time of the error. If the FSP does not implement this CallerId shall be zero (0).</td>
</tr>
<tr>
<td>ErrorType</td>
<td>A GUID identifying the nature of the fatal error. This GUID is platform specific. A listing of all possible GUIDs shall be provided by the Integration Guide.</td>
</tr>
<tr>
<td>Status</td>
<td>A code describing the error encountered. Please see section 12.2 for a listing of possible error codes.</td>
</tr>
</tbody>
</table>

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

11.1 ACPI

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

11.2 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.

11.3 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.
The declarations/definitions provided here were derived from the EDK2 source available for download at https://github.com/tianocore/edk2

12.1  **BOOT_MODE**

12.1.1  **PiBootMode.h**

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

```c
#define BOOT_WITH_FULL_CONFIGURATION 0x00
#define BOOT_WITH_MINIMAL_CONFIGURATION 0x01
#define BOOT_ASSUME_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
```
12.2 EFI_STATUS

12.2.1 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
#define EFI_COMPROMISED_DATA 0x80000021

typedef UINT64 EFI_PHYSICAL_ADDRESS;

12.2.2 OEM Status Code

The range of status code that has 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.
#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
12.3  EFI_PEI_GRAPHICS_INFO_HOB

12.3.1  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;

12.4  EFI_PEI_GRAPHICS_DEVICE_INFO_HOB

12.4.1  GraphicsInfoHob.h

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

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

12.5  EFI_GUID

12.5.1  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;
12.5.2 UefiBaseType.h

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

typedef GUID EFI_GUID;
12.6 EFI_MEMORY_TYPE

12.6.1 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;
12.7 Hand Off Block (HOB)

12.7.1 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    0x00100000
#define EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTABLE      0x00200000
#define EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTABLE      0x00200000
#define EFIRESOURCE_ATTRIBUTE_EXECUTION_PROTECTABLE 0x00400000
#define EFIRESOURCE_ATTRIBUTE_READ_ONLY_PROTECTABLE 0x00800000
#define EFIRESOURCE_ATTRIBUTE_PERSISTABLE 0x01000000
#define EFIRESOURCE_ATTRIBUTE_MORE_RELIABLE 0x02000000

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

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

/// Describes various attributes of logical memory allocation.
///
typedef struct {
    EFI_GUID         Name;
    EFI_PHYSICAL_ADDRESS MemoryBaseAddress;
    UINT64           MemoryLength;
    EFI_MEMORY_TYPE  MemoryType;
    UINT32           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;
12.8 Firmware Volume and Firmware Filesystem

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

12.8.1 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;

12.8.2 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_FV_FILETYPE_FREEFORM 0x02

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

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;
12.9 Debug Error Level

Please refer to DebugLib.h from the EDK2 project for the original source.

12.9.1 DebugLib.h

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

// Declare bits for PcdDebugPrintErrorLevel and the ErrorLevel parameter of DebugPrint()

#define DEBUG_INIT 0x00000001 // Initialization
#define DEBUG_WARN 0x00000002 // Warnings
#define DEBUG_LOAD 0x00000004 // Load events
#define DEBUG_FS 0x00000008 // EFI File system
#define DEBUG_POOL 0x00000010 // Alloc & Free (pool)
#define DEBUG_PAGE 0x00000020 // Alloc & Free (page)
#define DEBUG_INFO 0x00000040 // Informational debug messages
#define DEBUG_DISPATCH 0x00000080 // PEI/DXE/SMM Dispatchers
#define DEBUG_VARIABLE 0x00000100 // Variable
#define DEBUG_BM 0x00000400 // Boot Manager
#define DEBUG_BLKIO 0x00001000 // BlkIo Driver
#define DEBUG_NET 0x00004000 // Network Io Driver
#define DEBUG_UNDI 0x00010000 // UNDI Driver
#define DEBUG_LOADFILE 0x00020000 // LoadFile
#define DEBUG_EVENT 0x00080000 // Event messages
#define DEBUG_GCD 0x00100000 // Global Coherency Database changes
#define DEBUG_CACHE 0x00200000 // Memory range cachability changes
#define DEBUG_VERBOSE 0x00400000 // Detailed debug messages that may significantly impact boot performance
#define DEBUG_ERROR 0x80000000 // Error
// Aliases of debug message mask bits
//
#define EFI_D_INIT DEBUG_INIT
#define EFI_D_WARN DEBUG_WARN
#define EFI_D_LOAD DEBUG_LOAD
#define EFI_D_FS DEBUG_FS
#define EFI_D_POOL DEBUG_POOL
#define EFI_D_PAGE DEBUG_PAGE
#define EFI_D_INFO DEBUG_INFO
#define EFI_D_DISPATCH DEBUG_DISPATCH
#define EFI_D_VARIABLE DEBUG_VARIABLE
#define EFI_D_BM DEBUG_BM
#define EFI_D_BLKIO DEBUG_BLKIO
#define EFI_D_NET DEBUG_NET
#define EFI_D_UNDI DEBUG_UNDI
#define EFI_D_LOADFILE DEBUG_LOADFILE
#define EFI_D_EVENT DEBUG_EVENT
#define EFI_D_VERBOSE DEBUG_VERBOSE
#define EFI_D_ERROR DEBUG_ERROR

12.10 Event Code Types

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

typedef UINT32 EFI_STATUS_CODE_TYPE;
#define EFI_STATUS_CODE_TYPE_MASK 0x000000FF
#define EFI_STATUS_CODE_SEVERITY_MASK 0xFF000000
#define EFI_STATUS_CODE_RESERVED_MASK 0x00FFFF00
#define EFI_PROGRESS_CODE 0x00000001
#define EFI_ERROR_CODE 0x00000002
#define EFI_DEBUG_CODE 0x00000003
#define EFI_ERROR_MINOR 0x40000000
#define EFI_ERROR_MAJOR 0x80000000
#define EFI_ERROR_UNRECOVERED 0x90000000
#define EFI_ERROR_UNCONTAINED 0xA0000000

typedef UINT32 EFI_STATUS_CODE_VALUE;
#define EFI_STATUS_CODE_CLASS_MASK 0xFF000000
#define EFI_STATUS_CODE_SUBCLASS_MASK 0x00FF0000
#define EFI_STATUS_CODE_OPERATION_MASK 0x0000FFFF
#define EFI_SOFTWARE 0x03000000

#pragma warning(disable:4291)
12.11 **EFI_STATUS_CODE_STRING_DATA**

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

```c
#define EFI_STATUS_CODE_DATA_TYPE_STRING_GUID \
    { 0x92D11080, 0x496F, 0x4D95, \
        { 0xBE, 0x7E, 0x03, 0x74, 0x88, 0x38, 0x2B, 0x0A }}

typedef struct {
    UINT16 HeaderSize;
    UINT16 Size;
    EFI_GUID Type;
} EFI_STATUS_CODE_DATA;

typedef enum {
    EfiStringAscii,
    EfiStringUnicode,
    EfiStringToken
} EFI_STRING_TYPE;

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

typedef struct {
    EFI_STATUS_CODE_DATA           DataHeader;
    EFI_STRING_TYPE                StringType;
    EFI_STATUS_CODE_STRING         String;
} EFI_STATUS_CODE_STRING_DATA;
```
## Appendix B – Acronyms

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