Intel® Firmware Support Package

External Architecture Specification

April 2014

Document Number: 330554-001

Contents

Revision History

Date	Revision	Description
April 2014	001	Initial release.

§

1 Introduction

1.1 Purpose

The purpose of this document is to describe the external architecture and interfaces provided in the Intel® Firmware Support Package (FSP).

1.2 Intended Audience

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

1.3 Related Documents

- *Platform Initialization (PI) Specification* located at http://www.uefi.org/specifications
- *Binary Configuration Tool for Intel® Firmware Support Package* – available at http://www.intel.com/fsp

1.4 Conventions

To illustrate some of the points better, the document may use code snippets. The code snippets follow **the GNU C Compiler** and **GNU Assembler** syntax.

1.5 Acronyms and Terminology

BCT	Binary Configuration Tool
BSP	Boot Strap Processor
BSF	Boot Setting File
BWG	BIOS Writer's Guide. Interchangeable with FWG.
CRB	Customer Reference Board
FSP	Firmware Support Package
FSP API	Firmware Support Package Interface
FWG	Firmware Writer's Guide. Interchangeable with BWG.
HOB	Hand-Off-Block
IVI	In Vehicle Infotainment
NBSP	Node BSP
RSM	Resume to Operating System (OS) from SMM
PCH	Platform Controller Hub
SBSP	System BSP
SMI	System Management Interrupt
SMM	System Management Mode
TSEG	Memory Reserved at the Top of Memory to be used as SMRAM
TXE	Trusted Execution Engine/Environment
UPD	Updatable Product Data
VPD	Vital Product Data

2 FSP Overview

2.1 Design Philosophy

Intel recognizes that it holds the key programming information that is crucial for initializing Intel silicon. After Intel provides the key information, most experienced firmware engineers can make the rest of the system work by studying manuals and specifications.

FSP is a binary distribution of the silicon reference code that is needed to initialize the Intel silicon. The design goal of Intel Firmware Support Package (FSP) is to abstract the complexities of initialization of Intel Silicon and expose a limited number of well-defined interfaces.

2.2 Technical Overview

The Intel® Firmware Support Package (FSP) provides chipset and processor initialization in a format that can easily be incorporated into many existing boot loaders.

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

FSP is not a stand-alone boot loader; therefore it needs to be integrated into a host boot loader to carry out other boot loader functions, such as: initializing non-Intel components, conducting bus enumeration, and discovering devices in the system and all industry standard and Intel architectural initialization.

3 FSP Integration

The FSP binary can be integrated easily into many different boot loaders and also into the embedded OS directly.

Below are some required steps for the integration:

- **Customizing**

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

- **Rebasing**

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

- **Placing**

  Once the FSP binary is ready for integration, the boot loader build process needs to be modified to place this FSP binary at the specific rebasing location identified above.

- **Interfacing**

  The boot loader needs to add code to set up 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 for each hardware platform contains the following:

- FSP Binary
- Reference code that illustrates the interfacing mechanism required by the boot loader.
- VPD/UPD Data structure definitions
- BSF File
- Integration Guide

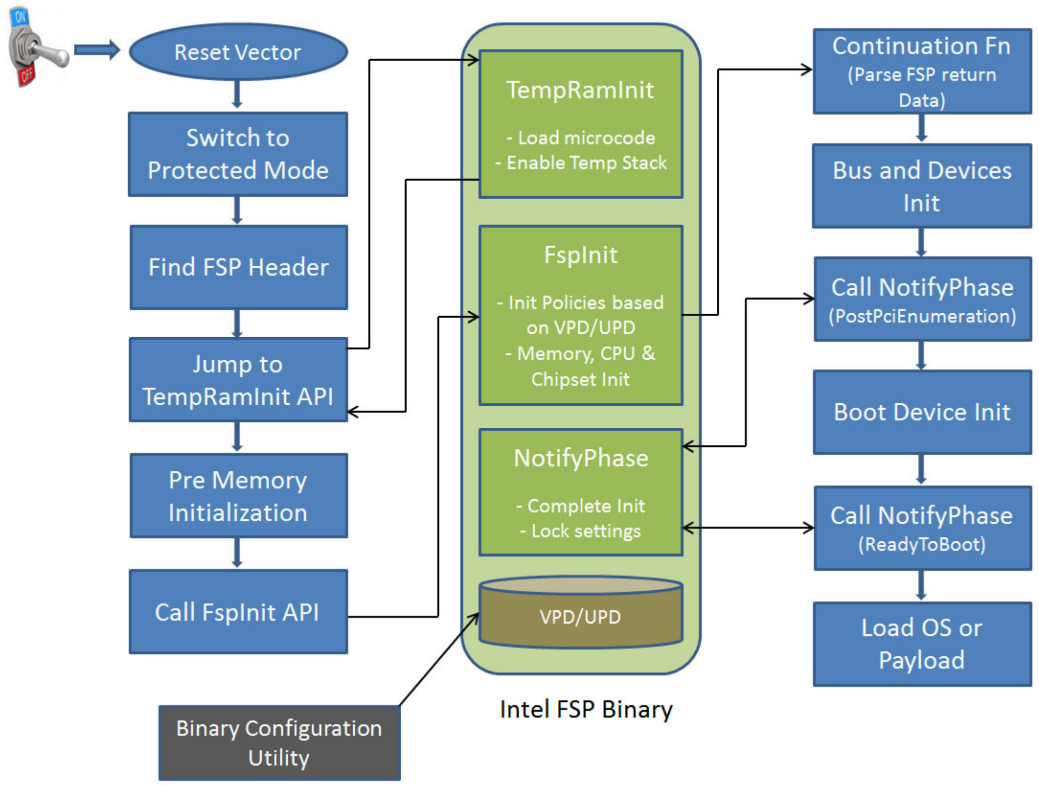
The FSP configuration utility called BCT is available as a separate package.

3.2 FSP Image ID and Revision

The FSP information header contains an Image Identifier field and an Image Revision field that provide the identification and revision information of the FSP binary. It is important to verify these fields while integrating the FSP as the FSP configuration data could change between different FSP Image identifiers and revisions.

4 Boot Flow

The figure below shows the boot flow from the reset vector to the OS handoff for a typical boot loader. The APIs are described in more detail in the following sections.



5 FSP Binary Format

The FSP is distributed in binary format. The FSP binary contains an FSP-specific **FSP_INFORMATION_HEADER** structure, the initialization code/data needed by the Intel Silicon supported by the FSP and a configuration region that allows the boot loader developer to customize some of the settings through the Binary Configuration Tool (BCT) provided by Intel.

5.1 FSP Header

The FSP header conveys the information required by the boot loader to interface with the FSP binary, such as providing the addresses for the entry points, configuration region address, etc.

Byte Offset	Size in Bytes	Field	Description
0	4	Signature	'FSPH'. Signature for the FSP information header.
4	4	HeaderLength	Length of the header
8	3	Reserved	Reserved
11	1	HeaderRevision	Revision of the header
12	4	ImageRevision	Revision of the FSP binary. The ImageRevision can be decoded as follows: 0..7 - Minor Version 8..15 - Major Version 16..31 - Reserved
16	8	Image Id	8-byte signature string that will help match the FSP binary to a supported hardware configuration.
24	4	ImageSize	Size of the entire FSP binary.
28	4	ImageBase	FSP binary preferred base address. If the FSP binary will be located at the address different from the preferred address, the rebasing tool is required to relocate the base before the FSP binary integration.
32	4	ImageAttribute	Attributes of the FSP binary. This field is not currently used.
36	4	CfgRegionOffset	Offset of the configuration region. This offset is relative to the FSP binary base address.
40	4	CfgRegionSize	Size of the configuration region.

Byte Offset	Size in Bytes	Field	Description
44	4	ApiEntryNum	Number of API entries this FSP supports. The current design supports three APIs as given below.
48	4	TempRamInitEntryOffset	The offset for the API to setup a temporary stack till the memory is initialized.
52	4	FspInitEntryOffset	The offset for the API to initialize the CPU and the chipset (SOC).
56	4	NotifyPhaseEntryOffset	The offset for the API to inform the FSP about the different stages in the boot process.
60	4	Reserved	Reserved

5.1.1 Finding the FSP Header

The FSP binary follows the *UEFI Platform Initialization Firmware Volume Specification* format. The Firmware Volume (FV) format is described in the *Platform Initialization (PI) Specification - Volume 3: Shared Architectural Elements* specification and can be downloaded from http://www.uefi.org/specifications.

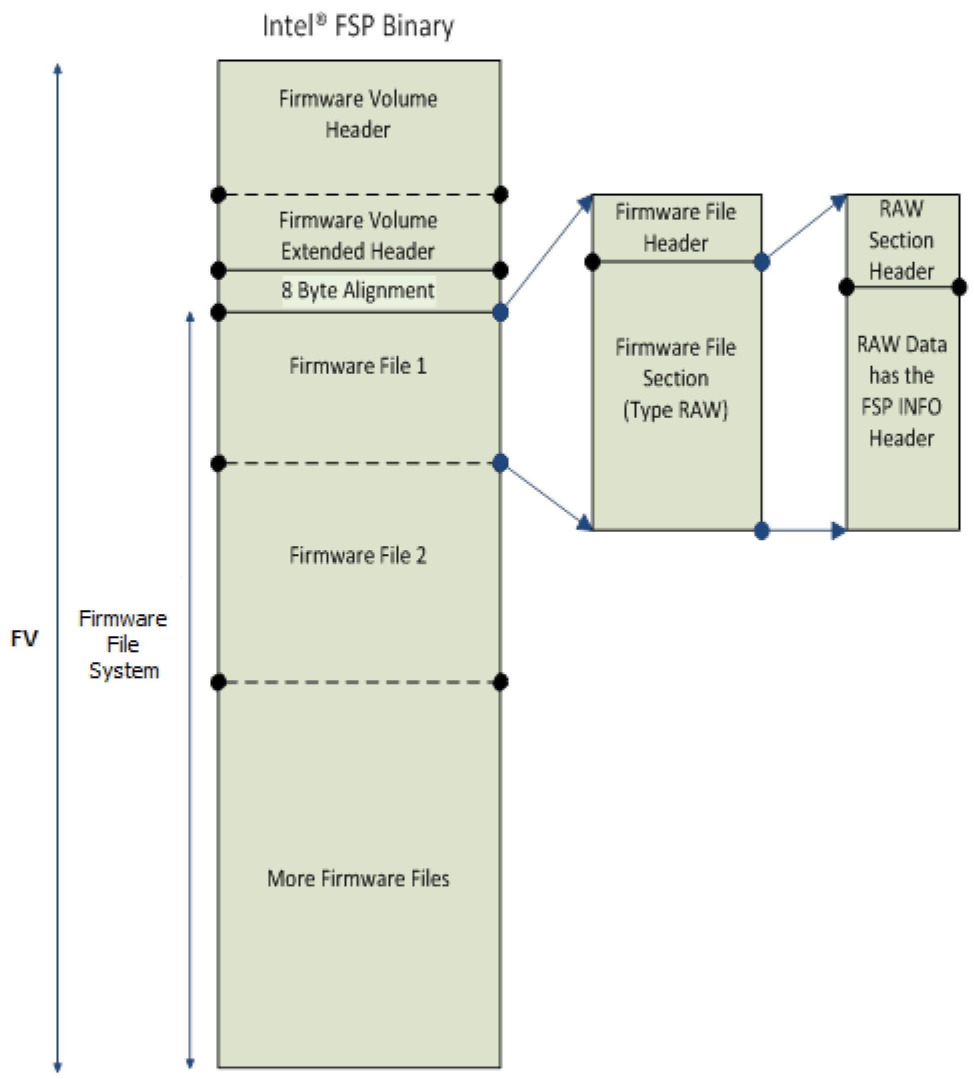
FV is a way to organize/structure binary components. It enables a standardized way to parse the binary and handle the individual binary components that make up the FV.

The FSP_INFORMATION_HEADER is a firmware file and is placed as the **first** firmware file within the firmware volume. All firmware files will have a GUID that can be used to identify the files, including the FSP Header file. The FSP header firmware file GUID is defined as **912740BE-2284-4734-B971-84B027353F0C**.

The boot loader can find the offset of the FSP header within the FSP binary by the following steps described below:

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

The next figure shows a pictorial representation of the data structures that is parsed in the above flow.



5.1.2 FSP Header Offset

To simplify the integration of the FSP binary with a boot loader, the offset of the FSP header will be provided with the FSP binary documentation. In this case, the boot loader may choose to skip the generic algorithm to find the FSP header as described above, but instead use the hardcoded value for the FSP header offset. This approach is easier to implement from the boot loader side.

For the FSP binary the FSP information header structure is placed at offset **0x94.**

6 FSP Interface (FSP API)

6.1 Entry-Point Calling Assumptions

There are some requirements regarding the operating environment for FSP execution. The boot loader is responsible to set up this operating environment before calling the FSP API. These conditions have to be met before calling any entry point or 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 4-GB access range.
- Interrupts should be turned off.
- The FSP API should be called only by the system BSP, unless otherwise noted.

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

6.2 Data Structure Convention

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

6.3 Entry-Point Calling Convention

All FSP APIs defined in the FSP information 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 are preserved.

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

6.4 Exit Convention

The TempRamInit API preserves all general purpose registers except EAX, ECX, and EDX. Because this FSP API is executing in a stackless environment, the floating point registers may be used by the FSP to save/restore other general purpose registers to the boot loader.

The FspInit and the FspNotify interfaces will preserve all the general purpose registers except EAX. The return status will be passed back through the EAX register.

The FSP reserves some memory for its internal use and the memory region that is used by the FSP is passed back though a hand off block (HOB). This is a generic resource HOB, but the owner field of the HOB will identify the owner as FSP. Refer to “FSP Output” in Section 7 for more details. The boot loader should not use this memory except for parsing the HOB output. The boot loader should also mark this memory as reserved when passing the memory map to the OS.

6.5 Return Status Code

All FSP APIs will return a status code to indicate the API execution result. FSP reuses a subset of the standard status codes defined in the EDK II specifications. See status codes listed below.

```
#define FSP_SUCCESS                0x00000000
#define FSP_INVALID_PARAMETER      0x80000002
#define FSP_UNSUPPORTED            0x80000003
#define FSP_NOT_READY              0x80000006
#define FSP_DEVICE_ERROR           0x80000007
#define FSP_OUT_OF_RESOURCES       0x80000009
#define FSP_VOLUME_CORRUPTED       0x8000000A
#define FSP_NOT_FOUND              0x8000000E
#define FSP_TIMEOUT                0x80000012
#define FSP_ABORTED                0x80000015
#define FSP_INCOMPATIBLE_VERSION   0x80000010
#define FSP_SECURITY_VIOLATION     0x8000001A
#define FSP_CRC_ERROR              0x8000001B
```

6.6 TempRamInitEntry

This FSP API is called soon after coming out of reset and before memory and stack is available. This FSP API will load the microcode update, enable code caching for the region specified by the boot loader and also setup a temporary stack to be used until main memory is initialized.

A hardcoded stack can be set up with the following values, and the “esp” register initialized to point to this hardcoded stack.

1. The return address where the FSP will return control after setting up a temporary stack.
2. A pointer to the input parameter structure

However, since the stack is in ROM and not writeable, this FSP API cannot be called using the “call” instruction, but needs to be jumped to.

This API should be called only once after the system comes out the reset, and it must be called before any other FSP APIs. The system needs to go through a reset cycle before this API can be called again. Otherwise, unexpected results may occur.

6.6.1 Prototype

```
typedef

FSP_STATUS

(FSPAPI *FSP_TEMP_RAM_INIT) (

  IN  FSP_TEMP_RAM_INIT_PARAMS        *TempRamInitParamPtr

);
```

6.6.2 Parameters

TempRaminitParamPtr

Address pointer to the **FSP_TEMP_RAM_INIT_PARAMS** structure. The structure definition is provided below under Related Definitions. The structure has a pointer to the base of a code region and the size of it. The FSP enables code caching for this region. Enabling code caching for this region should not take more than one MTRR pair. The structure also has a pointer to a microcode region and its size. The microcode region may have multiple microcodes packed together one after the other and the FSP will try to load all the microcodes that it finds in the region that is compatible with the silicon it is supporting. This microcode region is remembered by FSP so that it can be used to load microcode for all APs later on during the FspInit API call.

6.6.3 Related Definitions

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

`MicrocodeRegionBase`	Base address of the microcode region.
`MicrocodeRegionLength`	Length of the microcode region.
`CodeRegionBase`	Base address of the cacheable flash region.
`CodeRegionLength`	Length of the cacheable flash region.

6.6.4 Return Values

If this function is successful, the FSP initializes the **ECX and EDX** registers to point to a temporary but writeable memory range available to the boot loader and returns with FSP_SUCCESS in register EAX. Register ECX points to the start of this temporary memory range and EDX points to the end of the range. Boot loader is free to use the whole range described. Typically the boot loader can reload the ESP register to point to the end of this returned range so that it can be used as a standard stack.

Note: This returned range is just a sub-region of the whole temporary memory initialized by the FSP. The FSP maintains and consumes the remaining temporary memory. The boot loader must not access the temporary memory beyond the returned boundary.

FSP_SUCCESS	Temp RAM was initialized successfully.
FSP_INVALID_PARAMETER	Input parameters are invalid.
FSP_NOT_FOUND	No valid microcode was found in the microcode region.
FSP_UNSUPPORTED	The FSP calling conditions were not met.
FSP_DEVICE_ERROR	Temp RAM initialization failed.

6.6.5 Description

The entry to this function is in a stackless/memoryless environment. After the boot loader completes its initial steps, it finds the address of the FSP INFO HEADER and then from the header finds the offset of the TempRamInit function. It then converts the offset to an absolute address by adding the base of the FSP binary and jumps to the TempRamInit function.

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

The FSP will initialize the ECX and EDX registers to point to a temporary but writeable memory range. Register ECX points to the start of this temporary memory range and EDX points to the end of the range. The size of the temporary stack for the platform can be calculated by taking the range between ECX and EDX.

6.7 FspInitEntry

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

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

6.7.1 Prototype

```
typedef

FSP_STATUS

(FSPAPI *FSP_FSP_INIT) (

  INOUT  FSP_INIT_PARAMS        *FspInitParamPtr

);
```

6.7.2 Parameters

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

6.7.3 Related Definitions

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

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

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

`Status`	Status of the FSP INIT API.
`HobBufferPtr`	Pointer to the HOB data structure defined in the PI specification.

```
typedef struct {
  UINT32                          *StackTop;
  UINT32                           BootMode;
  VOID                            *UpdDataRgnPtr;
  UINT32                           Reserved[7];
} FSP_INIT_RT_COMMON_BUFFER;
```

```
typedef struct {
  FSP_INIT_RT_COMMON_BUFFER      Common;
  …
} FSP_INIT_RT_BUFFER;
```

`StackTop`	Point to the desired boot loader stack top location in memory after memory is initialized.
`BootMode`	Current boot mode. Refer to sample code file fsp_bootmode.h for the definitions.
`UpdDataRgnPtr`	Pointer to an updatable platform configuration data structure `UPD_DATA_REGION` defined in sample code file fsp_vpd.h. This structure contains options that can be overridden by the bootloader at runtime. If this pointer is `NULL,` it indicates the default built-in values in the FSP binary will be used. Refer to Section 8 for details.
`Reserved`	Reserved fields. Must be set to 0.

6.7.4 Return Values

FSP_SUCCESS	FSP execution environment was initialized successfully.
FSP_INVALID_PARAMETER	Input parameters are invalid.
FSP_UNSUPPORTED	The FSP calling conditions were not met.
FSP_DEVICE_ERROR	FSP initialization failed.

6.7.5 Description

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

ContinuationFunc is a function entry point that will be jumped to at the end of the FspInit() to transfer control back to the boot loader.

Note that this FspInit API initializes the permanent memory and switches the stack from the temporary memory to the permanent memory as specified by **`StackTop`**. Sometimes switching the stack in a function can cause some unexpected execution results because the compiler is not aware of the stack change during runtime and the precompiled code may still refer to the old stack for data and pointers. A stack switch therefore requires assembly code to go patch the data for the new stack location which may lead to compatibility issues.

To avoid such possible compatibility issues introduced by different compilers and to ease the integration of FSP with a boot loader, the API uses the **`ContinuationFunction`** parameter to continue the boot loader execution flow rather than return as a normal C function. Although this API is called as a normal C function, it never returns.

The FSP needs to get some parameters from the boot loader when it is initializing the silicon. These parameters are passed from the boot loader to the FSP through the **`FSP_INIT_RT_BUFFER`** structure pointer. Refer to the related FSP integration guide for the detailed structure definitions.

A set of parameters that the FSP may need to initialize memory under special circumstances, such as during an S3 resume and during fast boot mode, are returned by the FSP to the boot loader during a normal boot. The boot loader is expected to store these parameters in non-volatile memory, such as in the SPI flash, and return a pointer to this structure (through **`NvsBufferPtr`**) when it is requesting the FSP to initialize the silicon under these special circumstances. Refer to Section 7.3 for the details on how to get the returned NVS data from FSP.

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

6.8 NotifyPhaseEntry

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

- Post PCI enumeration
- Ready To Boot

6.8.1 Prototype

```
typedef
FSP_STATUS
(FSPAPI *FSP_NOTFY_PHASE) (
  IN  NOTIFY_PHASE_PARAMS        *NotifyPhaseParamPtr
);
```

6.8.2 Parameters

NotifyPhaseParamPtr	Address pointer to the `NOTIFY_PHASE_PRAMS`

6.8.3 Related Definitions

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

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

`EnumInitPhaseAfterPciEnumeration`

This stage is notified when the boot loader completed the PCI enumeration and the resource allocation for the PCI devices is complete. FSP will use it to do some specific initialization for processor and chipset that requires PCI resource assignment.

`EnumInitPhaseReadyToBoot`

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

6.8.4 Return Values

FSP_SUCCESS	The notification was handled successfully.
FSP_UNSUPPORTED	The notification was not called in the proper order.
FSP_INVALID_PARAMETER	The notification code is invalid.

6.8.5 Description

The FSP will lock the configuration registers to enhance security as required by the BIOS Writer's Guide (BWG)/BIOS Specification when it is notified that the boot loader is ready to transfer control to the operating system.

Therefore, this API should only be called after the FspInit API and each notification code should be called only once in the predefined order. For example, the **`EnumInitPhaseAfterPciEnumeration`** notification needs to be called before the **`EnumInitPhaseReadyToBoot`** notification. Once the **`EnumInitPhaseReadyToBoot`** is notified, the whole FSP flow is considered to be completed and no further FSP API calls are allowed.

7 FSP Output

The FSP builds a series of data structures called the Hand-Off-Blocks (HOBs) as it progresses through initializing the silicon. These data structures conform to the HOB format as described in the *Platform Initialization (PI) Specification - Volume 3: Shared Architectural Elements* specification and can be downloaded from http://www.uefi.org/specifications

The user of the FSP binary is strongly encouraged to go through the specification mentioned above to understand the HOB design details and create a simple infrastructure to parse the HOBs, because the same infrastructure can be reused with different FSPs across different platforms

The boot loader developer must decide on how to consume the information passed through the HOBs produced by the FSP. For example, even the specification mentioned above describes about nine different HOBs; most of this information may not be relevant to a particular boot loader. For example, a boot loader design may be interested only in knowing the amount of memory populated and may not care about any other information.

The section below describes the GUID HOBs that are produced by the FSP. GUID HOB structures are non-architectural in the sense that the structure the HOB needs is not defined in the HOB specifications. So the GUID and the data structure are documented below to enable the boot loader to consume these HOB data.

7.1 Boot Loader Temporary Memory Data HOB

As described in the FspInit API, the system memory is initialized and the whole temporary memory is destroyed during this API call. However, the sub region of the temporary memory returned in the TempRamInit API may still contain boot loader-specific data which might be useful for the boot loader even after the FspInit call. So before destroying the temporary memory, all contents in this sub region will be migrated to the permanent memory, FSP builds a boot loader temporary memory data HOB and the boot loader can use it to access the data saved in the temporary memory after FspInit API if necessary. If the boot loader does not care about the previous data in the stack, this HOB can be simply ignored.

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

```
#define FSP_BOOTLOADER_TEMPORARY_MEMORY_HOB_GUID \

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

To retrieve this HOB data, refer to GetBootloaderTempMemoryBuffer () function in the sample file fsp_support.c which illustrates how to get the boot loader temporary memory back from the FSP HOB.

7.2 FSP Reserved Memory Resource Descriptor HOB

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

```
#define FSP_HOB_RESOURCE_OWNER_FSP GUID \

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

To retrieve this HOB data, refer to GetFspReservedMemory() function in the sample file fsp_support.c, which illustrates how to get the FSP reserved memory from the HOB.

7.3 Non-Volatile Storage HOB

```
#define FSP_NON_VOLATILE_STORAGE_HOB_GUID \

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

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

The boot loader needs to parse the HOB list to see if such a GUID HOB exists after returning from the FspInit() API. If so, the boot loader should extract the data portion from the HOB, and then save it into a platform-specific NVS device, such as flash, EEPROM, etc. The next time the system boots, the boot loader should load the data block back from the NVS device to temporary memory and populate the buffer pointer into FSP_INIT_PARAMS.NvsBufferPtr field before calling into the FspInit() API. If the NVS device is memory mapped, the boot loader can initialize the buffer pointer directly to the buffer.

To retrieve this HOB data, refer to the function implementation of GetFspNvsDataBuffer () in sample file fsp_support.c for a code snippet that illustrates how to get the FSP NVS data that needs to be saved by the boot loader.

8 FSP Configuration Firmware File

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

The configurable data region has two sets of data

VPD – Vital Product Data, which can only be configured statically.

UPD – Updatable Product Data, which can be configured statically for default values, but also can be overridden during boot at runtime.

Both the VPD and the UPD parameters can be statically customized using a separate tool called the Binary Configuration Tool (BCT) as explained in the tools section. The tool uses a Boot Setting File (BSF) to understand the layout of the configuration region within the FSP.

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

When calling the FspInit API, the stack is in temporary memory where the UPD data structure is copied, updated, and passed to the FSP API. When permanent memory is initialized, the FSP sets up a new stack in the permanent memory and tears down the temporary memory. However, the FSP saves the whole boot loader temporary memory region in a GUID HOB. If the boot loader needs to access the old data in the temporary memory, it can be done by parsing the HOB to retrieve the previous temporary memory data. Note that the migrated temporary memory contains an identical copy of the original data. If pointers are stored in this region, they need to be fixed to point to the new migrated region before they are used.

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

9 Tools

The Binary Configuration Tool (BCT) is available at http://www.intel.com/fsp, which allows a user to modify certain well defined configuration values in the FSP binary. The BCT provides a Graphical User Interface (GUI) for changing these configuration values. The BCT includes separate documentation that explains the usage of the tool. Refer to Section 1.3 for the BCT documentation information.

10 Other Host Boot Loader Concerns

10.1 Power Management

Intel® FSP does not provide power management functions besides making power management features available to the host boot loader. ACPI is an independent component of the boot loader, and it is not included in the Intel® FSP.

10.2 Bus Enumeration

Intel® FSP initializes the CPU and the companion chips to a state where all bus topology can be discovered by the host boot loader.

10.3 Security

The FSP follows the BWG / BIOS Specification to set the necessary registers for security concerns. However, some other security features, such as secure boot, are not covered by the current FSP. If the secure boot feature is required, contact your local Intel representative.

10.4 Pre-OS Graphics

The current FSP binary does not include the graphics initialization function. For other pre-OS graphics initialization solutions, contact your local Intel representative.