

**Intelligent
Systems**

Intel[®] Firmware Support Package

Introduction Guide

October 2012



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, the Intel logo, and Intel Atom are trademarks of Intel Corporation in the U.S. and/or other countries.

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

Copyright © 2012, Intel Corporation. All rights reserved.



Contents

1.0	Introduction	5
1.1	Firmware and Boot Loader Functions Overview	5
1.2	Intelligent Systems and Embedded Ecosystem Overview	5
2.0	Scope	5
3.0	Intel® FSP Overview	6
3.1	Design Philosophy	6
3.2	Technical Overview	6
3.3	The Creation of Intel® FSP	6
3.4	Intel® FSP Integration	7
3.4.1	Boot Flow	8
3.4.2	Locate Intel® FSP	9
3.4.3	Intel® FSP Header	9
3.4.4	Intel® FSP Functions Invocation	12
3.4.5	Entry-Point Calling Assumptions	12
3.4.6	Entry-Point Calling Convention	12
3.4.7	Exit Convention	12
3.4.8	Intel® FSP Hand-Off Data to Host Boot Loader	20
3.5	Customize Intel® FSP Internal Configuration	22
3.6	Rebase Intel® FSP	22
4.0	Other Host Boot Loader Concerns	23
4.1	SMM	23
4.2	S3 Resume	23
4.3	Fastboot Option	23
4.4	Power Management	23
4.5	Bus Enumeration	23
4.6	Security	23
4.7	64-bit Long Mode	23
4.8	Pre-OS Graphics	23
5.0	Glossary	24
A	HOB Parsing Sample Code	24
B	Sample Code to Find Intel® FSP Header	42

Figures

1	Intel® FSP Creation and Usage Model	7
2	Intel® FSP Boot Flow	8
3	Intel® FSP Header Information	10
4	Intel® FSP Binary Internal Structure	11



Revision History

Date	Revision	Description
October 2012	001	Initial release



1.0 Introduction

1.1 Firmware and Boot Loader Functions Overview

Regardless of the microprocessor architecture, firmware has always been an essential part of the system software that is responsible for hardware abstraction to present the programming interface to the upper layer software.

Since Intel Architect has been traditionally tied to PC architecture from the debut of the IBM PC in 1981, the firmware layer, named “Basic Input and Output System” (BIOS) has been a de facto standard for the industry.

With the evolution of PC architecture, BIOS has grown from a simple 16-bit programming interface to become an agent that:

- initializes the hardware,
- presents hardware configuration, and
- provides run-time services to software and operating systems.

With the introduction of Unified Extensible Firmware Interface (UEFI) a decade ago, the modern 64-bit firmware standard has gradually replaced the aging legacy BIOS. This improves its efficiency and eliminates some constraints that hinder the growth of PC technology, such as Hard Disk size limit. UEFI inherits the same roles as the BIOS.

BIOS and UEFI are often called “boot loader,” interchangeably, but the term “boot loader” is frequently used in the embedded applications. “Boot loader” is frequently referred to as the firmware that runs from the microprocessor reset vector to the stage where the first piece of kernel code of an operating system starts.

Even though BIOS or UEFI can be used in embedded applications, there are many choices of boot loader for a system under design, such as Real-Time Operating System (RTOS), Linux, Open Source firmware, etc. Besides the differences in purposes of these options, there is one common function among them: to initialize the hardware underneath and provide a programming interface to the upper layer software.

1.2 Intelligent Systems and Embedded Ecosystem Overview

Contrasting to the PC ecosystem where hardware and software architecture are following a set of industry standards, the Intelligent Systems (embedded) ecosystem often does not adhere to the same industry standards. Design engineers for Intelligent Systems and Embedded Systems frequently combine components from different vendors with a set of very distinct functions in mind. The criteria for picking the right boot loader are often based on boot speed and code size. The boot loader also frequently has close ties with the OS from a functionality perspective. To give freedom to customers to choose the best boot loader for their applications, Intel provides the Intel® Firmware Support Package (Intel® FSP) to satisfy the needs of design engineers.

2.0 Scope

This document describes the usage guidelines of the Intel® FSP, the programming interface of the Intel® FSP, and sample implementation.

The document is intended for the following audience:

- Firmware/boot loader engineers and architects.
- Firmware solutions providers and enablers.



3.0 Intel® FSP Overview

3.1 Design Philosophy

Intel recognizes that Intel 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 specifications, porting guides, and reference code.

Intel® Firmware Support Package does just that. It contains crucial code for initializing Intel microprocessors and their companion chips. A firmware engineer can easily integrate Intel® FSP into a boot loader of their choice, and all they have to do is to integrate other initialization code for other platform components and implement the features required by the platform.

3.2 Technical Overview

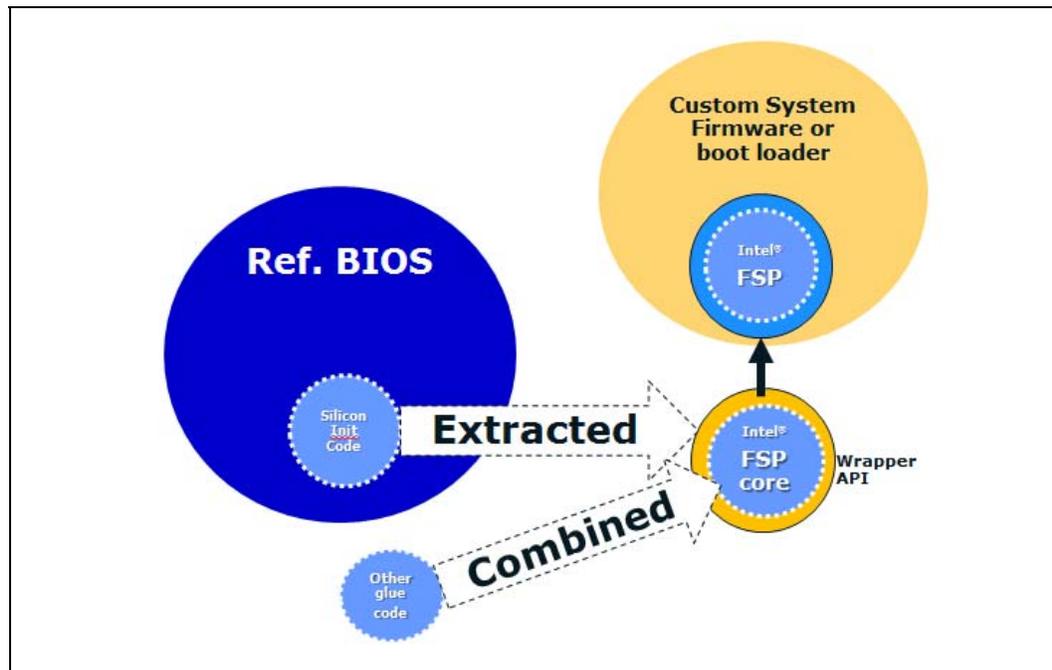
Intel® Firmware Support Package (Intel® FSP) provides basic CPU and companion chip initialization functions in a form that can be easily integrated into any boot loader used in an embedded application.

Intel® FSP performs all the necessary initialization steps as documented in the BIOS Writers' Guide (BWG), including initialization of the CPU, memory controller, chipset and certain bus interfaces, if necessary. Intel® 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.

3.3 The Creation of Intel® FSP

Intel® FSP is created from the existing UEFI based firmware. After the initialization modules are validated in the PEIM form in a reference BIOS or an equivalent boot loader for a reference platform, the code is extracted and packaged into a Intel® FSP FV (Firmware Volume) for distribution. The Intel® FSP binary can then be integrated into a boot loader following the programming interface and design guidelines.

Figure 1. Intel® FSP Creation and Usage Model



3.4 Intel® FSP Integration

The Intel® FSP binary can be integrated easily into many different boot loaders, such as Coreboot, VxWorks, BIOS, etc.

Below are some required steps for the integration:

- Customizing
The static Intel® FSP configuration parameters are part of the Intel® FSP binary and can be customized by external tools that will be provided by Intel.
- Rebasing
The Intel® FSP is not Position Independent Code (PIC) and the whole Intel® FSP has to be rebased if it is placed at a location which is different from the preferred address specified during building the Intel® FSP.
- Placing
After the Intel® FSP binary is ready for integration, the boot loader build process needs to be modified to place this Intel® 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 Intel® FSP, call the Intel® FSP with the correct parameters, and parse the Intel® FSP output to retrieve the necessary information returned by the Intel® FSP.

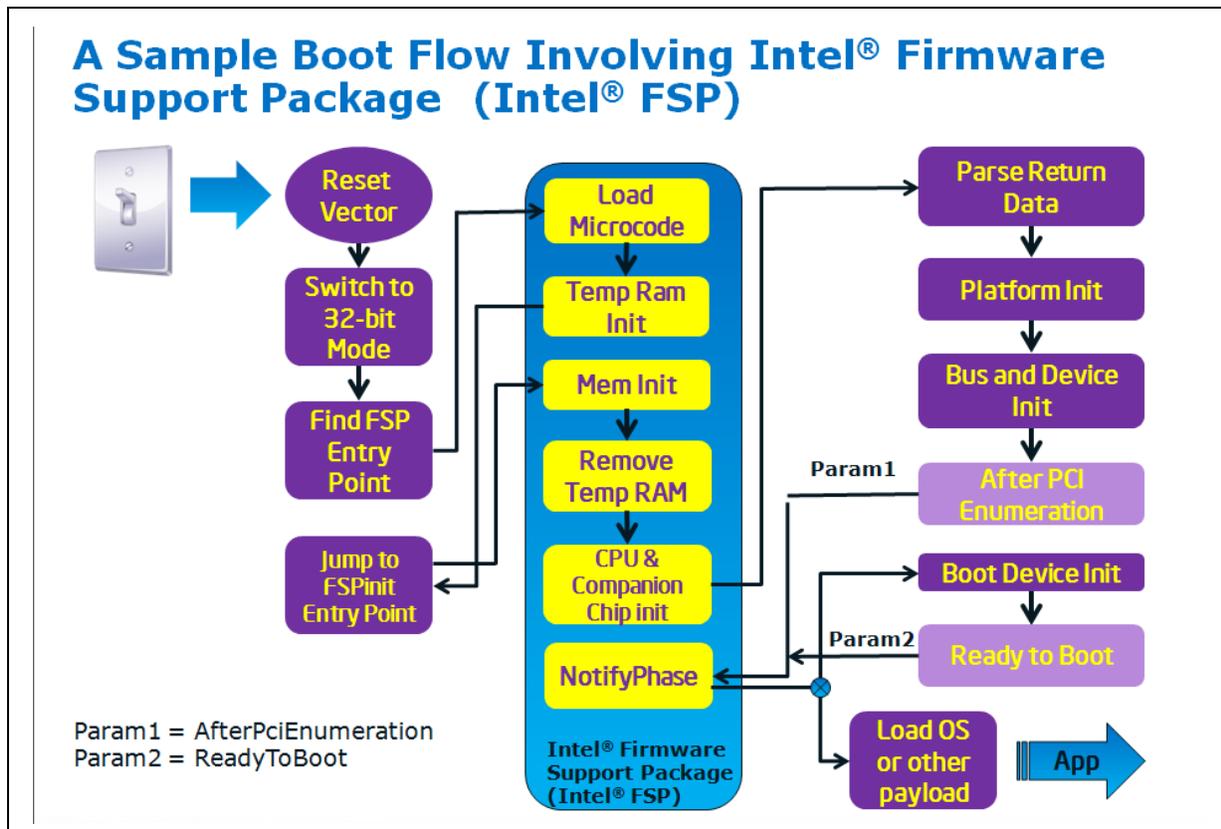
3.4.1 Boot Flow

As shown in Figure 1, Intel® FSP is not a stand-alone boot loader, it is designed to be integrated into a host boot loader, such as a RTOS, BIOS, Coreboot, U-Boot, or any proprietary boot loader.

The execution flow is also shown in Figure 2. After power is applied to the system, the execution (or instruction pointer) starts at the “Reset Vector,” the host boot loader is responsible for switching the boot mode into 32-bit mode, then finds the entry point of Intel® FSP. The host boot loader prepares a couple of things, and then jump into Intel® FSP. When Intel® FSP returns, it passes some system data back to the host boot loader to store or process, such as memory size and other important parameters that the boot loader might want to know.

After Intel® FSP hands off the control back to the host boot loader, the host boot loader will continue to initialize the rest of the system, and execute code to carry out features of the platform.

Figure 2. Intel® FSP Boot Flow





3.4.2 Locate Intel® FSP

3.4.2.1 Intel® FSP Binary Format

The Intel® FSP binary follows the UEFI Platform Initialization Firmware Volume Specification format. The firmware volume (FV) is described in the *Volume 3: Shared Architecture Elements* specification. For developers who are not familiar with the terminology, here are some quick definitions and explanations:

- A *firmware device* is a persistent physical repository that contains firmware code and/or data. It is typically a flash component but may be some other type of persistent storage.
- A single physical firmware device may be divided into smaller pieces to form multiple logical firmware devices.
- Similarly, multiple physical firmware devices may be aggregated into one larger logical firmware device.
- A logical firmware device is called a *firmware volume*.

After Intel® FSP is integrated into the host boot loader and forms an integrated binary, the details of FV and its data structure can be ignored and you can access Intel® FSP directly via a hardcoded value.

3.4.2.2 Intel® FSP Binary Integration Requirement

Intel® FSP binary will have a default location (e.g., 0xFFF80000) for integration with a host boot loader. Intel will provide a tool to “rebase” the location based on the need of the host boot loader. The default location will be documented in the Integration Guide for each Intel® FSP release.

3.4.2.3 Discovery of Intel® FSP Binary Entry Point

The Intel® FSP is distributed in binary format. It contains an Intel® FSP-specific FSP_INFORMATION_HEADER structure, the initialization code, and data needed by the Intel silicon, and a configuration region that allows the boot loader developer to customize some of the settings through a tool provided by Intel.

3.4.3 Intel® FSP Header

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



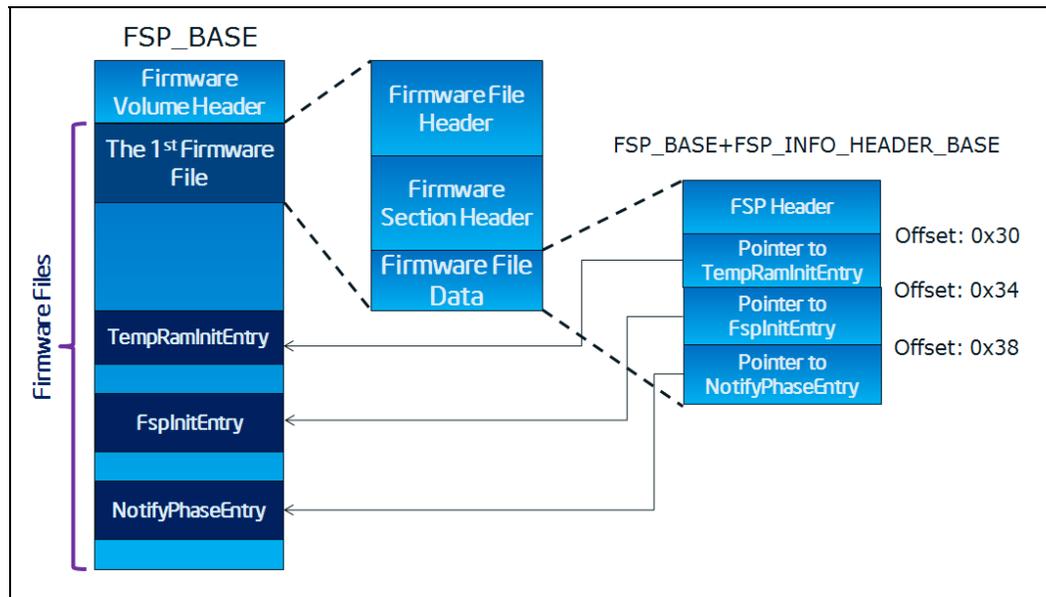
Figure 3. Intel® FSP Header Information

Byte Offset	Size in Bytes	Field	Description
0	4	Signature	'FSPH'. Signature for the Intel® 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 Intel® FSP Binary.
16	8	Image Id	8-byte signature string that will help match the Intel® FSP Binary to a supported hardware configuration.
24	4	ImageSize	Size of the entire Intel® FSP Binary.
28	4	ImageBase	Intel® FSP binary preferred base address. If the Intel® FSP binary will be located at the address different from the preferred address, the rebasing tool is required to relocate the base before the Intel® FSP binary integration.
32	4	ImageAttribute	Attributes of the Intel® FSP binary.
36	4	CfgRegionOffset	Offset of the configuration region. This offset is relative to the Intel® FSP binary base address.
40	4	CfgRegionSize	Size of the configuration region.
44	4	ApiEntryNum	Number of API Entries this Intel® FSP supports. The current design supports 3 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 Intel® FSP about the different stages in the boot process.

3.4.3.1 Finding the Intel® FSP Header

By the time the host boot loader is ready to find Intel® FSP, the state of the microprocessor is this: no cache, no stack, and no memory. The host boot loader should have brought the microprocessor from reset mode to a 32-bit mode with 4GB address range.

Figure 4. Intel® FSP Binary Internal Structure



3.4.3.1.1 The “Easy” Way

The “Easiest” way to find whether the Intel® FSP is using hardcoded values:

- The Intel® FSP header locates at `FSP_BASE+0x94`
- The pointer to Intel® FSP's “TempRamInitEntry” locates at an offset of 0x30 from the Intel® FSP header. The pointer to Intel® FSP's “FspInitEntry” locates at an offset of 0x34 from the Intel® FSP header.
- The pointer to Intel® FSP's “NotifyPhaseEntry” locates at an offset of 0x38 from the Intel® FSP header.

All these hardcoded values are default values for the Intel® FSP with which they are associated and each release of Intel® FSP may have different default values; the developer can “rebase” the Intel® FSP to somewhere else without impacting the offsets of these values. The “rebase” tool will be a topic for a later explanation.

3.4.3.1.2 The “Hard” Way

The more standard way to find Intel® FSP is to walk through the Firmware Volume data structure:

1. Find the base of the Intel® FSP FV header, which is at `0xFFF80000` unless it is “rebased.”
2. Use `EFI_FIRMWARE_VOLUME_HEADER` to parse the Intel® FSP FV header, and skip the standard and extended FV header.
3. Locate the `EFI_FFS_FILE_HEADER`.
4. Then, from the header data structure, locate the `EFI_RAW_SECTION` header.
5. Locate the data section in the Raw section, and find the Intel® FSP INFORMATION HEADER there.
6. From `FSP_INFORMATION_HEADER`, find each programming interface and entry points.



Because the steps need to be performed before memory is available, special care must be taken to avoid C code compiler issues associated with stack assumptions. Please refer to sample code in [Appendix B](#) to study details.

3.4.4 Intel® FSP Functions Invocation

3.4.5 Entry-Point Calling Assumptions

There are some requirements regarding the operating environment for Intel® FSP execution. It is the responsibility of the boot loader to set up this operating environment before calling the FSPI. These conditions have to be met before calling any entry point or the behavior is not determined. These conditions include:

- 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 FSPI should be called only by the System BSP, unless otherwise noted.

Other requirements needed by individual FSPI will be covered in their respective sections.

3.4.6 Entry-Point Calling Convention

- All three Intel® FSP Interfaces follow the default C `__cdecl` calling convention. This implies that the stack is cleaned up by the caller.
 - The `TempRamInit` API needs special handling as explained in the section about it.
- All three Intel® FSP Interfaces take a pointer to a structure as an input parameter.

3.4.7 Exit Convention

- All three Intel® FSP Interfaces return an unsigned 32 bit integer as status.
- Intel® FSP guarantees that the “ebp” register will be preserved during `TempRamInit` Intel® FSP interface. Since this Intel® FSP interface is executing in a stackless environment, no other register is guaranteed to be preserved.
- The `FspInit` and `FspNotify` interfaces will preserve all registers except “eax” as the return status will be passed back through it.
- The Intel® FSP will reserve some memory for its internal use and the boot loader is expected to not to use this memory except to parse the HOB output. The boot loader is also expected to mark this memory as reserved when constructing the memory map information to be passed to the OS.

As mentioned previously, before calling any of the Intel® FSP functions, the host boot loader must be in 32-bit mode, with data and code selectors having 4GB addressing range. All interrupts also should be disabled.

All Intel® FSP APIs follow the default `__cdecl` calling convention; therefore the caller needs to clean up the stack.



3.4.7.1 After the invocation of Intel® FSP Functions

All Intel® FSP functions return an unsigned 32-bit integer as status.

Intel® FSP reserves a memory region (currently it is 1MB in size) for its own use throughout the operation of the system.

3.4.7.2 TempRamInitEntry

This FSPI is called soon after coming out of reset and before memory and stack are available.

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

- a. The return address where the Intel® FSP will return control after setting up a temporary stack
- b. A pointer to the input parameter structure

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

3.4.7.2.1 Prototype

```
typedef
FSP_STATUS
(FSPAPI *FSP_TEMP_RAM_INIT) (
    IN FSP_TEMP_RAM_INIT_PARAM    *TempRamInitParamPtr
);
```

3.4.7.2.2 Parameters

MtrrRegionPtr

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 Intel® FSP binary in memory and the size of the Intel® FSP binary. This is used by the Intel® FSP to enable code caching for the Intel® FSP. 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 Intel® FSP will try to load all the microcodes that it finds in the region that is compatible with the silicon it is supporting.

3.4.7.2.3 Related Definitions

```
typedef struct {
    UINT32    MicrocodeRegionBase,
    UINT32    MicrocodeRegionLength,
    UINT32    FspBinaryBase,
    UINT32    FspBinaryLength
} FSP_TEMP_RAM_INIT_PARAMS;
```



3.4.7.2.4 Return Values

FSP_SUCCESS	Temp RAM was initialized successfully.
FSP_INVALID_PARAMETER	Input parameters are invalid.

3.4.7.2.5 Sample Code

```
.equ  FSP_BIN_BASE,          0xFFF80000

.global basic_init
basic_init:
    .
    .
    .

    #
    # Parse the FV to find the FSP INFO Header
    #
    lea  findFspHeaderStack, %esp
    jmp  find_fsp_entry
findFspHeaderDone:
    mov  %eax, %ebp          # save fsp header address in
ebp
    mov  0x30(%ebp), %eax
    add  0x1c(%ebp), %eax

    lea  tempRamInitStack, %esp    # initialize to a rom stack

    #
    # call FSP PEI to setup temporary Stack
    #
    jmp  *%eax

temp_RamInit_done:
    addl $4, %esp

    #
    # call C based function to initialize memory and chipset. Pass
the FSP INFO
    # Header address as a parameter
    #
    .
    .
    .
```



```

#
# should never return here
#
jmp .

.align 4
findFspHeaderStack:
.long findFspHeaderDone

tempRamInitParams:
.long _ucode_base      # Microcode base address
.long _ucode_size     # Microcode size
.long 0xffff80000     # Code Region Base
.long 0x00040000     # Code Region Length

tempRamInitStack:
.long temp_RamInit_done # return address
.long tempRamInitParams # pointer to parameters

```

3.4.7.2.6 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 Intel® 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 Intel® FSP binary and calls the TempRamInit function.

Before return, the Intel® FSP would initialize the ESP register to a point to the top of a temporary but writeable stack. Now that a stack is available, the boot loader can switch to using C. This temporary stack should be used to do only the minimal initialization that needs to be done before memory can be initialized.

3.4.7.3 FspInitEntry

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

3.4.7.3.1 Prototype

```

typedef
FSP_STATUS
(FSPAPI *FSP_FSP_INIT) (
    INOUT FSP_INIT_PARAMS    *FspInitParamPtr
);

```



3.4.7.3.2 Parameters

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

3.4.7.3.3 Related Definitions

```
typedef struct {
    VOID *NvsBufferPtr;
    VOID *RtBufferPtr;
    VOID *HobBufferPtr;
} FSP_INIT_PARAMS;
```

NvsBufferPtr Pointer to the non-volatile storage data buffer.
RtBufferPtr Pointer to the runtime data buffer.
HobBufferPtr Pointer to the HOB data structure defined in the PI specification.

3.4.7.3.4 Return Values

FSP_SUCCESS	Intel® FSP execution environment was initialized successfully.
FSP_INVALID_PARAMETER	Input parameters are invalid.

3.4.7.3.5 Sample Code

```
typedef struct {
    void *NvsBufferPtr;
    void *RtBufferPtr;
    void *HobBufferPtr;
} FSP_INIT_PARAMS;

typedef struct {
    UINT32 *StackTop;
} FSP_INIT_RT_COMMON_BUFFER;

typedef struct {
    FSP_INIT_RT_COMMON_BUFFER Common;
} FSP_INIT_RT_BUFFER;

void boot_loader_rom_stage_fn (FSP_INFORMATION_HEADER
*fsp_info)
{
    .
    .
    .
}
```



```

uint32_t                                FspInitEntry;
volatile FSP_INIT_PARAMS                 FspInitParams;
volatile FSP_INIT_RT_BUFFER              FspRtBuffer;

memset((void*)&FspRtBuffer, 0, sizeof(FSP_INIT_RT_BUFFER));
FspRtBuffer.Common.StackTop = &_amp;_stack_top;
FspInitParams.NvsBufferPtr = 0;
FspInitParams.HobBufferPtr = 0;
FspInitParams.RtBufferPtr = (FSP_INIT_RT_BUFFER *)&FspRtBuffer;
FspInitEntry = fsp_info->ImageBase + fsp_info->FspInitEntry;

/*
When FSP returns, the temporary stack is migrated to the RAM @
_stack_top. Because ebp will still be pointing to the old temp
_stack, _ebp needs to be patched to enable access to local vari-
ables.
The logic is to find the offset from the stack pointer that ebp
is pointing to before the stack switch and apply the same off-
set after the stack switch.
*/

/* call FSP PEI to setup MRC and other CS init */

    "movl %0, (%esp)\n\t" /* Save the FspInitParams ptr in
the stack */
    "movl %%esp, %eax\n\t" /* Save the current esp in ebx */
    "subl %%eax, %%ebp\n\t" /* Find the offset from the stack
top and ebp */
    "movl %1, %eax\n\t" /* Move the FSP Init API address
into eax */
    "call %%eax \n\t" /* Call FSP Init API */
    "addl %%esp, %%ebp\n\t" /* Add the new stack top to the
offset in ebp */
    :: "g"(&FspInitParams), "g"(FspInitEntry): "eax");

/*

Since the stack switch happened, the leave instruction cannot be
used as it will reset the ESP. So don't return from this function
without fixing the esp stored in the stack. The other easier
alternative is to just jump out and not return to this function.

It is generally advisable to jump out of this function instead of
returning from it, because to return from each level, the esp must
be patched in the stack frame for that level.

*/

```



```
/*  
  
Call boot_loader_ram_stage_fn to complete system initialization.  
boot_loader_ram_stage_fn should NOT return.  
  
*/  
    boot_loader_ram_stage_fn (fsp_info, FspInitParams.Hob-  
        BufferPtr);  
    dead_loop();  
}  
  
boot_loader_ram_stage_fn (FSP_INFORMATION_HEADER *fsp_info,  
    void *FspHobListPtr);  
{  
    .  
    .  
    .  
}
```

3.4.7.4 NotifyPhaseEntry

This FSPI is used to notify the Intel® FSP about the different phases in the boot process. This allows the Intel® FSP to take appropriate actions as needed during different initialization phases. The phases will be platform dependent and will be documented with the Intel® FSP release. Examples of boot phases include “post PCI enumeration” and “ready to boot.”

The Intel® FSP will lock the configuration registers to enhance security as required by the BWG when it is notified that the boot loader is ready to transfer control to the operating system. Because the configuration registers may have to be locked on multiple processor cores, the boot loader has to notify the Intel® FSP about the “ready to boot” phase individually from each of the processor threads that is enabled in the system. It is the responsibility of the boot loader to serialize this call as the Intel® FSP is not expected to support reentrancy.

3.4.7.4.1 Prototype

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

3.4.7.4.2 Parameters

NotifyPhaseParamPtr Address pointer to the **NOTIFY_PHASE_PARAMS**



3.4.7.4.3 Related Definitions

```
typedef enum {
    EnumInitPhaseAfterPciEnumeration,
    EnumInitPhaseReadyToBoot
} 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. Intel® FSP uses it to do some specific initialization for the processor and chipset that requires PCI resource assignment.

EnumInitPhaseReadyToBoot

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

3.4.7.4.4 Return Values

FSP_SUCCESS	Always returns success.
-------------	-------------------------

3.4.7.4.5 Sample Code

```
#define FSPAPI __attribute__((cdecl))

typedef UINT32 FSP_STATUS;
typedef FSP_STATUS (FSPAPI *FSP_NOTIFY_PHASE)
(NOTIFY_PHASE_PARAMS *NotifyPhaseParamPtr);

typedef enum {
    EnumInitPhaseAfterPciEnumeration,
    EnumInitPhaseReadyToBoot
} FSP_INIT_PHASE;
```



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

void boot_loader_post_pci_notify_fn ()
{
    FSP_NOTIFY_PHASE    NotifyPhaseProc;
    NOTIFY_PHASE_PARAMS    NotifyPhaseParams;

    NotifyPhaseParams.Phase = EnumInitPhaseAfterPciEnumeration;

    /* call FSP to Notify PostPciEnumeration */
    /* fsp_info_header is a global variable and is a pointer to
the */
    /* FSP INFORMATION_HEADER and is assumed to be initialized
before this
    /* func is called */
    NotifyPhaseProc = (FSP_NOTIFY_PHASE ) (fsp_info_header-
>ImageBase + fsp_info_header->NotifyPhaseEntry);
    NotifyPhaseProc (&NotifyPhaseParams);
}
}
```

3.4.8 Intel® FSP Hand-Off Data to Host Boot Loader

Intel® FSP builds a couple of data structures as it goes through its initialization process. It passes them to the host boot loader when it exits.

The data structures comply with UEFI Hand-Off-Block (HOB) format described in *Volume 3: Shared Architectural Elements* specification, which can be downloaded from <http://www.uefi.org/specs/>.

It's left to the boot loader developer to decide on how to consume the information passed through the HOBs produced by the Intel® 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. With this in mind, some sample code is provided below which parses the HOBs and finds the amount of memory populated in the system.

The HOB infrastructure code is embedded as files and an example function using that infrastructure and getting the memory information is provided in the code form.



Here is a sample code that retrieves memory size information from HOB.

```

void
GetMemorySize (
    uint32_t          *LowMemoryLength,
    void              *HobBufferPtr
)
{
    EFI_PEI_HOB_POINTERS    Hob;

    *LowMemoryLength = 0x100000;

    //
    // Get the HOB list for processing
    //
    Hob.Raw = HobBufferPtr;

    //
    // Collect memory ranges
    //
    while (!END_OF_HOB_LIST (Hob)) {
        if (Hob.Header->HobType ==
            EFI_HOB_TYPE_RESOURCE_DESCRIPTOR) {
            if (Hob.ResourceDescriptor->ResourceType ==
                EFI_RESOURCE_SYSTEM_MEMORY) {
                //
                // Need memory above 1MB to be collected here
                //
                if (Hob.ResourceDescriptor->PhysicalStart >= 0x100000 &&
                    Hob.ResourceDescriptor->PhysicalStart <
                    (EFI_PHYSICAL_ADDRESS) 0x100000000) {
                    *LowMemoryLength += (uint32_t) (Hob.ResourceDescrip-
                        tor->ResourceLength);
                }
            }
        }
        Hob.Raw = GET_NEXT_HOB (Hob);
    }

    return;
}

void boot_loader_rom_stage_fn ()

```



```
{  
    .  
    .  
    .  
    /* call FSP PEI to setup MRC and other CS init */  
    FspInitApi = (FSP_FSP_INIT) (*(uint32_t *) (FSP_INIT_API));  
  
    asm ("movl %0, (%esp)\n\t" /* Save the FspInitParams ptr in  
the stack */  
        "movl %1, %%eax\n\t" /* Move FSP Init API address  
into eax */  
        "movl %%esp, %%ebx\n\t" /* Save the current esp in ebx  
*/  
        "subl %%ebx, %%ebp\n\t" /* Find the offset from the  
stack top */  
        "call %%eax \n\t" /* Call FSP Init API */  
        "addl %%esp, %%ebp\n\t" /* Add the offset to the new  
stack top */  
        : : "g"(&FspInitParams), "g"(FspInitApi));  
  
    /* Get the memory size by parsing the HOB returned from the  
FSP */  
    GetMemorySize (&LowMemoryLength, FspInitParams.HobBufferPtr);  
    .  
    .  
    .  
}
```

3.5 Customize Intel® FSP Internal Configuration

Intel® FSP contains a configurable data region used by Intel® FSP during initialization. Typically this is the default parameters used by the PEIMs in the Intel® FSP. The parameters can be statically customized using a separate tool called Binary Configuration Tool (BCT). The tool will examine the Binary Setting File (BSF) to understand the layout of the configuration region within the Intel® FSP, and allow a user to modify certain well defined configuration values in the binary. The BCT tool will be provided with each Intel® FSP release.

3.6 Rebase Intel® FSP

During integration of Intel® FSP and the host boot loader, the developer needs to place the Intel® FSP binary at a default physical location based on the Integration Guide of a specific Intel® FSP release.

If the default address needs to be relocated, the developer needs to use a rebase tool provided by Intel to relocate the binary to another location in the ROM. The tool will be released with the 1st official release of Intel® FSP.



4.0 Other Host Boot Loader Concerns

4.1 SMM

SMM code will be provided only if there is a silicon workaround that necessitates it. It is the responsibility of the host boot loader to provide the necessary infrastructure in terms of rendezvousing all the processors and calling the SMM code provided by Intel® FSP in SMM mode. Details about the SMM handler may be platform/silicon dependent.

4.2 S3 Resume

The host boot loader can detect the boot mode (e.g., warm boot, cold boot, and S3 resume); therefore it is the responsibility of the host boot loader to notify Intel® FSP about the boot mode by passing the flag in the NVS Buffer. Details will be provided later.

4.3 Fastboot Option

Intel® FSP will check a flag in the RT_Buffer structure to decide if the memory training algorithm can be skipped in order to speed up the boot process. This option is available only after the memory training code has been run at least once and the parameters of memory training result are preserved.

4.4 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 will not be included in Intel® FSP.

4.5 Bus Enumeration

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

4.6 Security

Intel® FSP locks registers based on Intel's recommendation in Intel documents. Other system-level security features need to be done by the host boot loader.

4.7 64-bit Long Mode

Intel® FSP operates in 32-bit mode; it is the responsibility of the host boot loader to transition to 64-bit Long Mode if desired.

4.8 Pre-OS Graphics

Intel® FSP does not include graphics initialization function. For pre-OS graphics initialization solutions, please contact your Intel representative.



5.0 Glossary

BIOS	Basic Input and Output System
BSP	Boot Strap Processor
BWG	BIOS Writer's Guide
CRB	Customer Reference Board
EDK	EFI Development Kit
DXE	Drive eXecution Environment
EFI	Extensible Firmware Interface
FSP	Intel® Firmware Support Package
FWG	Firmware Writer's Guide
IVI	In Vehicle Infotainment
PEI	Pre-EFI Initialization
PEIM	PEI Module
SMM	System Management Mode
TSEG	Memory Reserved at the Top of Memory to be used as SMRAM
UEFI	Unified Extensible Firmware Interface

Appendix A HOB Parsing Sample Code

The sample code provided here was derived from the EDK2 source available for download at

<http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK2>

```
///
/// 8-byte unsigned value.
///
typedef unsigned long long  UINT64;
///
/// 8-byte signed value.
///
typedef long long          INT64;
///
/// 4-byte unsigned value.
///
typedef unsigned int      UINT32;
///
/// 4-byte signed value.
///
typedef int               INT32;
///
/// 2-byte unsigned value.
///
```



```

typedef unsigned short      UINT16;
///
/// 2-byte Character. Unless otherwise specified all strings
are stored in the
/// UTF-16 encoding format as defined by Unicode 2.1 and ISO/
IEC 10646 standards.
///
typedef unsigned short      CHAR16;
///
/// 2-byte signed value.
///
typedef short               INT16;
///
/// Logical Boolean. 1-byte value containing 0 for FALSE or a
/// 1 for TRUE. Other values are undefined.
///
typedef unsigned char       BOOLEAN;
///
/// 1-byte unsigned value.
///
typedef unsigned char       UINT8;
///
/// 1-byte Character
///
typedef char                CHAR8;
///
/// 1-byte signed value
///
typedef char                INT8;

typedef void                VOID;

typedef UINT64              EFI_PHYSICAL_ADDRESS;

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

#define CONST      const
#define STATIC     static

#define TRUE      ((BOOLEAN) (1==1))
#define FALSE    ((BOOLEAN) (0==1))

```



```
static inline void DebugDeadLoop(void) {
    for (;;)
}

#define FSPAPI __attribute__((cdecl))
#define EFIAPI __attribute__((cdecl))

#define _ASSERT(Expression) DebugDeadLoop()
#define ASSERT(Expression) \
do { \
    if (!(Expression)) { \
        _ASSERT (Expression); \
    } \
} while (FALSE)

typedef UINT32 FSP_STATUS;
typedef UINT32 EFI_STATUS;

//
// 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                0xFFFF
#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 {
    ///
    /// Identifies the HOB data structure type.
    ///
    UINT16    HobType;
    ///
    /// The length in bytes of the HOB.
    ///
    UINT16    HobLength;
    ///
    /// This field must always be set to zero.
    ///
    UINT32    Reserved;
} EFI_HOB_GENERIC_HEADER;
```



```
///
/// Enumeration of memory types introduced in UEFI.
///
typedef enum {
    ///
    /// Not used.
    ///
    EfiReservedMemoryType,
    ///
    /// The code portions of a loaded application.
    /// (Note that UEFI OS loaders are UEFI applications.)
    ///
    EfiLoaderCode,
    ///
    /// The data portions of a loaded application and the default
    /// data allocation type used by an application to allocate
    pool memory.
    ///
    EfiLoaderData,
    ///
    /// The code portions of a loaded Boot Services Driver.
    ///
    EfiBootServicesCode,
    ///
    /// The data portions of a loaded Boot Services Driver, and the
    default data
    /// allocation type used by a Boot Services Driver to allocate
    pool memory.
    ///
    EfiBootServicesData,
    ///
    /// The code portions of a loaded Runtime Services Driver.
    ///
    EfiRuntimeServicesCode,
    ///
    /// The data portions of a loaded Runtime Services Driver and
    the default
    /// data allocation type used by a Runtime Services Driver to
    allocate pool memory.
    ///
    EfiRuntimeServicesData,
    ///
    /// Free (unallocated) memory.
    ///
    EfiConventionalMemory,
    ///
    /// Memory in which errors have been detected.
```



```
///
EfiUnusableMemory,
///
/// Memory that holds the ACPI tables.
///
EfiACPIReclaimMemory,
///
/// Address space reserved for use by the firmware.
///
EfiACPIMemoryNVS,
///
/// Used by system firmware to request that a memory-mapped IO
region
/// be mapped by the OS to a virtual address so it can be
accessed by EFI runtime services.
///
EfiMemoryMappedIO,
///
/// System memory-mapped IO region that is used to translate
memory
/// cycles to IO cycles by the processor.
///
EfiMemoryMappedIOPortSpace,
///
/// Address space reserved by the firmware for code that is
part of the processor.
///
EfiPalCode,
EfiMaxMemoryType
} EFI_MEMORY_TYPE;

///
/// EFI_HOB_MEMORY_ALLOCATION_HEADER describes the
/// various attributes of the logical memory allocation. The
/// type field will be used for subsequent inclusion in the
/// UEFI memory map.
///
typedef struct {
    ///
    /// A GUID that defines the memory allocation region's type
    and purpose, as well as
    /// other fields within the memory allocation HOB. This GUID
    is used to define the
    /// additional data within the HOB that may be present for the
    memory allocation HOB.
    /// Type EFI_GUID is defined in InstallProtocolInterface() in
    the UEFI 2.0
    /// specification.
    ///
```



```

EFI_GUID                Name;

///
/// The base address of memory allocated by this HOB. Type
/// EFI_PHYSICAL_ADDRESS is defined in AllocatePages() in the
UEFI 2.0
/// specification.
///
EFI_PHYSICAL_ADDRESS    MemoryBaseAddress;

///
/// The length in bytes of memory allocated by this HOB.
///
UINT64                  MemoryLength;

///
/// Defines the type of memory allocated by this HOB. The mem-
ory type definition
/// follows the EFI_MEMORY_TYPE definition. Type
EFI_MEMORY_TYPE is defined
/// in AllocatePages() in the UEFI 2.0 specification.
///
EFI_MEMORY_TYPE        MemoryType;

///
/// Padding for Itanium processor family
///
UINT8                   Reserved[4];
} EFI_HOB_MEMORY_ALLOCATION_HEADER;

///
/// Describes all memory ranges used during the HOB producer
/// phase that exist outside the HOB list. This HOB type
/// describes how memory is used, not the physical attributes
of memory.
///
typedef struct {
    ///
    /// The HOB generic header. Header.HobType =
EFI_HOB_TYPE_MEMORY_ALLOCATION.
    ///
    EFI_HOB_GENERIC_HEADER    Header;
    ///
    /// An instance of the EFI_HOB_MEMORY_ALLOCATION_HEADER that
describes the
    /// various attributes of the logical memory allocation.
    ///
    EFI_HOB_MEMORY_ALLOCATION_HEADER    AllocDescriptor;
}

```



```
//
// Additional data pertaining to the "Name" Guid memory
// may go here.
//
} EFI_HOB_MEMORY_ALLOCATION;

///
/// The resource type.
///
typedef UINT32 EFI_RESOURCE_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

///
/// A type of recount attribute type.
///
typedef UINT32 EFI_RESOURCE_ATTRIBUTE_TYPE;

//
// 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

///
/// Describes the resource properties of all fixed,
/// nonrelocatable resource ranges found on the processor
/// host bus during the HOB producer phase.
///
typedef struct {
    ///
    /// The HOB generic header. Header.HobType =
    EFI_HOB_TYPE_RESOURCE_DESCRIPTOR.
    ///
    EFI_HOB_GENERIC_HEADER      Header;
    ///
    /// A GUID representing the owner of the resource. This GUID
    is used by HOB
    /// consumer phase components to correlate device ownership
    of a resource.
    ///
    EFI_GUID                    Owner;
    ///
    /// The resource type enumeration as defined by
    EFI_RESOURCE_TYPE.
    ///
    EFI_RESOURCE_TYPE           ResourceType;
    ///
    /// Resource attributes as defined by
    EFI_RESOURCE_ATTRIBUTE_TYPE.
    ///
    EFI_RESOURCE_ATTRIBUTE_TYPE ResourceAttribute;

```



```
///
/// The physical start address of the resource region.
///
EFI_PHYSICAL_ADDRESS      PhysicalStart;
///
/// The number of bytes of the resource region.
///
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 {
    ///
    /// The HOB generic header. Header.HobType =
EFI_HOB_TYPE_GUID_EXTENSION.
    ///
    EFI_HOB_GENERIC_HEADER      Header;
    ///
    /// A GUID that defines the contents of this HOB.
    ///
    EFI_GUID                    Name;
    //
    // Guid specific data goes here
    //
} 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;

/**
Returns the type of a HOB.

This macro returns the HobType field from the HOB header for
the
```



```

HOB specified by HobStart.

@param HobStart  A pointer to a HOB.

@return HobType.

**/
#define GET_HOB_TYPE(HobStart) \
  ((* (EFI_HOB_GENERIC_HEADER **) &(HobStart)) ->HobType)

/**
  Returns the length, in bytes, of a HOB.

  This macro returns the HobLength field from the HOB header for
  the
  HOB specified by HobStart.

  @param HobStart  A pointer to a HOB.

  @return HobLength.

**/
#define GET_HOB_LENGTH(HobStart) \
  ((* (EFI_HOB_GENERIC_HEADER **) &(HobStart)) ->HobLength)

/**
  Returns a pointer to the next HOB in the HOB list.

  This macro returns a pointer to HOB that follows the
  HOB specified by HobStart in the HOB List.

  @param HobStart  A pointer to a HOB.

  @return A pointer to the next HOB in the HOB list.

**/
#define GET_NEXT_HOB(HobStart) \
  (VOID *) (*(UINT8 **) &(HobStart) + GET_HOB_LENGTH (HobStart))

/**
  Determines if a HOB is the last HOB in the HOB list.

  This macro determine if the HOB specified by HobStart is the
  last HOB in the HOB list.  If HobStart is last HOB in the HOB
  list,
  then TRUE is returned.  Otherwise, FALSE is returned.

```



```
@param HobStart    A pointer to a HOB.

@retval TRUE       The HOB specified by HobStart is the last
HOB in the HOB list.
@retval FALSE      The HOB specified by HobStart is not the
last HOB in the HOB list.

**/
#define END_OF_HOB_LIST(HobStart) (GET_HOB_TYPE (HobStart) ==
(UINT16)EFI_HOB_TYPE_END_OF_HOB_LIST)

/**
Returns a pointer to data buffer from a HOB of type
EFI_HOB_TYPE_GUID_EXTENSION.

This macro returns a pointer to the data buffer in a HOB spec-
ified by HobStart.
HobStart is assumed to be a HOB of type
EFI_HOB_TYPE_GUID_EXTENSION.

@param GuidHob    A pointer to a HOB.

@return A pointer to the data buffer in a HOB.

**/
#define GET_GUID_HOB_DATA(HobStart) \
(VOID *) (*(UINT8 **) &(HobStart) + sizeof (EFI_HOB_GUID_TYPE))

/**
Returns the size of the data buffer from a HOB of type
EFI_HOB_TYPE_GUID_EXTENSION.

This macro returns the size, in bytes, of the data buffer in
a HOB specified by HobStart.
HobStart is assumed to be a HOB of type
EFI_HOB_TYPE_GUID_EXTENSION.

@param GuidHob    A pointer to a HOB.

@return The size of the data buffer.

**/
#define GET_GUID_HOB_DATA_SIZE(HobStart) \
(UINT16) (GET_HOB_LENGTH (HobStart) - sizeof
(EFI_HOB_GUID_TYPE))

/**
Returns the pointer to the HOB list.

This function returns the pointer to first HOB in the list.
```



```

    If the pointer to the HOB list is NULL, then ASSERT().

    @return The pointer to the HOB list.

**/
VOID *
EFIAPI
GetHobList (
    VOID
);

/**
    Returns the next instance of a HOB type from the starting HOB.

    This function searches the first instance of a HOB type from
    the starting HOB pointer.
    If there does not exist such HOB type from the starting HOB
    pointer, it will return NULL.
    In contrast with macro GET_NEXT_HOB(), this function does not
    skip the starting HOB pointer
    unconditionally: it returns HobStart back if HobStart itself
    meets the requirement;
    caller is required to use GET_NEXT_HOB() if it wishes to skip
    current HobStart.

    If HobStart is NULL, then ASSERT().

    @param Type          The HOB type to return.
    @param HobStart      The starting HOB pointer to search from.

    @return The next instance of a HOB type from the starting HOB.

**/
VOID *
EFIAPI
GetNextHob (
    UINT16                Type,
    CONST VOID            *HobStart
);

/**
    Returns the first instance of a HOB type among the whole HOB
    list.

    This function searches the first instance of a HOB type among
    the whole HOB list.
    If there does not exist such HOB type in the HOB list, it will
    return NULL.

```



```
If the pointer to the HOB list is NULL, then ASSERT().

@param Type          The HOB type to return.

@return The next instance of a HOB type from the starting HOB.

**/
VOID *
EFIAPI
GetFirstHob (
    UINT16          Type
);

/**
    Returns the next instance of the matched GUID HOB from the
    starting HOB.

    This function searches the first instance of a HOB from the
    starting HOB pointer.
    Such HOB should satisfy two conditions:
    its HOB type is EFI_HOB_TYPE_GUID_EXTENSION and its GUID Name
    equals to the input Guid.
    If there does not exist such HOB from the starting HOB
    pointer, it will return NULL.
    Caller is required to apply GET_GUID_HOB_DATA () and
    GET_GUID_HOB_DATA_SIZE ()
    to extract the data section and its size info respectively.
    In contrast with macro GET_NEXT_HOB(), this function does not
    skip the starting HOB pointer
    unconditionally: it returns HobStart back if HobStart itself
    meets the requirement;
    caller is required to use GET_NEXT_HOB() if it wishes to skip
    current HobStart.

    If Guid is NULL, then ASSERT().
    If HobStart is NULL, then ASSERT().

    @param Guid          The GUID to match with in the HOB list.
    @param HobStart      A pointer to a Guid.

    @return The next instance of the matched GUID HOB from the
    starting HOB.

**/
VOID *
EFIAPI
GetNextGuidHob (
    CONST EFI_GUID   *Guid,
```



```

CONST VOID          *HobStart
);

/**
  Returns the first instance of the matched GUID HOB among the
  whole HOB list.

  This function searches the first instance of a HOB among the
  whole HOB list.
  Such HOB should satisfy two conditions:
  its HOB type is EFI_HOB_TYPE_GUID_EXTENSION and its GUID Name
  equals to the input Guid.
  If there does not exist such HOB from the starting HOB
  pointer, it will return NULL.
  Caller is required to apply GET_GUID_HOB_DATA () and
  GET_GUID_HOB_DATA_SIZE ()
  to extract the data section and its size info respectively.

  If the pointer to the HOB list is NULL, then ASSERT().
  If Guid is NULL, then ASSERT().

  @param  Guid          The GUID to match with in the HOB list.

  @return The first instance of the matched GUID HOB among the
  whole HOB list.

**/
VOID *
EFIAPI
GetFirstGuidHob (
  CONST EFI_GUID          *Guid
);

//
// Pointer to the HOB should be initialized with the output of
// FSP_INIT_PARAMS
//
extern volatile void *FspHobListPtr;

/**
  Reads a 64-bit value from memory that may be unaligned.

  This function returns the 64-bit value pointed to by Buffer.
  The function
  guarantees that the read operation does not produce an align-
  ment fault.

  If the Buffer is NULL, then ASSERT().

```



@param Buffer Pointer to a 64-bit value that may be unaligned.

@return The 64-bit value read from Buffer.

```
*/
UINT64
EFIAPI
ReadUnaligned64 (
    CONST UINT64          *Buffer
)
{
    ASSERT (Buffer != NULL);

    return *Buffer;
}
```

```
/**
    Compares two GUIDs.
```

This function compares Guid1 to Guid2. If the GUIDs are identical then TRUE is returned.

If there are any bit differences in the two GUIDs, then FALSE is returned.

If Guid1 is NULL, then ASSERT().

If Guid2 is NULL, then ASSERT().

@param Guid1 A pointer to a 128 bit GUID.

@param Guid2 A pointer to a 128 bit GUID.

@retval TRUE Guid1 and Guid2 are identical.

@retval FALSE Guid1 and Guid2 are not identical.

```
*/
BOOLEAN
EFIAPI
CompareGuid (
    CONST EFI_GUID *Guid1,
    CONST EFI_GUID *Guid2
)
{
    UINT64 LowPartOfGuid1;
    UINT64 LowPartOfGuid2;
    UINT64 HighPartOfGuid1;
    UINT64 HighPartOfGuid2;
```



```

    LowPartOfGuid1 = ReadUnaligned64 ((CONST UINT64*) Guid1);
    LowPartOfGuid2 = ReadUnaligned64 ((CONST UINT64*) Guid2);
    HighPartOfGuid1 = ReadUnaligned64 ((CONST UINT64*) Guid1 +
1);
    HighPartOfGuid2 = ReadUnaligned64 ((CONST UINT64*) Guid2 +
1);

    return (BOOLEAN) (LowPartOfGuid1 == LowPartOfGuid2 &&
HighPartOfGuid1 == HighPartOfGuid2);
}

/**
 Returns the pointer to the HOB list.
**/
VOID *
EFIAPI
GetHobList (
    VOID
)
{
    ASSERT (FspHobListPtr != NULL);
    return ((VOID *)FspHobListPtr);
}

/**
 Returns the next instance of a HOB type from the starting HOB.
**/
VOID *
EFIAPI
GetNextHob (
    UINT16                Type,
    CONST VOID            *HobStart
)
{
    EFI_PEI_HOB_POINTERS Hob;

    ASSERT (HobStart != NULL);

    Hob.Raw = (UINT8 *) HobStart;
    //
    // Parse the HOB list until end of list or matching type is
found.
    //
    while (!END_OF_HOB_LIST (Hob)) {
        if (Hob.Header->HobType == Type) {
            return Hob.Raw;
        }
    }
}

```



```
    }
    Hob.Raw = GET_NEXT_HOB (Hob);
  }
  return NULL;
}

/**
 Returns the first instance of a HOB type among the whole HOB
 list.
 **/
VOID *
EFIAPI
GetFirstHob (
    UINT16                Type
)
{
    VOID                *HobList;

    HobList = GetHobList ();
    return GetNextHob (Type, HobList);
}

/**
 Returns the next instance of the matched GUID HOB from the
 starting HOB.
 **/
VOID *
EFIAPI
GetNextGuidHob (
    CONST EFI_GUID        *Guid,
    CONST VOID            *HobStart
)
{
    EFI_PEI_HOB_POINTERS  GuidHob;

    GuidHob.Raw = (UINT8 *) HobStart;
    while ((GuidHob.Raw = GetNextHob
(EFI_HOB_TYPE_GUID_EXTENSION, GuidHob.Raw)) != NULL) {
        if (CompareGuid (Guid, &GuidHob.Guid->Name)) {
            break;
        }
        GuidHob.Raw = GET_NEXT_HOB (GuidHob);
    }
    return GuidHob.Raw;
}

/**
```



Returns the first instance of the matched GUID HOB among the whole HOB list.

```
**/  
VOID *  
EFIAPI  
GetFirstGuidHob (  
    CONST EFI_GUID      *Guid  
)  
{  
    VOID      *HobList;  
  
    HobList = GetHobList ();  
    return GetNextGuidHob (Guid, HobList);  
}
```



Appendix B Sample Code to Find Intel® FSP Header

The sample code provided below parses the Intel® FSP binary and finds the address of the Intel® FSP Header within it.

As the FV parsing has to be done before stack is available, a mix of assembly language code and C code is used. The C code is used to parse the data structures and find the Intel® FSP INFO Header. However, since the compiler will add Prolog or Epilog code to the C function, inline assembly is used to bypass those portions of the C code.

```
#include...
```

```
void __attribute__((optimize("O0"))) find_fsp_header ()
{
    volatile register UINT8 *ptr asm ("eax");

    __asm__ __volatile__ (
        ".global find_fsp_entry\n\t"
        "find_fsp_entry:\n\t"
    );

    //
    // Start at the FSP / FV Header base
    //
    ptr = (UINT8 *)0xFFF80000;

    //
    // Validate FV signature _FVH
    //
    if ((EFI_FIRMWARE_VOLUME_HEADER *)ptr)->Signature !=
0x4856465F) {
        ptr = 0;
        goto NotFound;
    }

    //
    // Add the Ext Header size to the Ext Header base to go to
the end of FV header
    //
    ptr += ((EFI_FIRMWARE_VOLUME_HEADER *)ptr)->ExtHeaderOff-
set;
    ptr += ((EFI_FIRMWARE_VOLUME_EXT_HEADER *)ptr)->ExtHeader-
Size;

    //
    // Align the end of FV header address to 8 bytes
    //
    ptr = (UINT8 *)(((UINTN)ptr + 7) & 0xFFFFFFFF8);

    //

```



```

// Now ptr is pointing to the FFS Header. Verify if the GUID
// matches the FSP_INFORMATION_HEADER GUID
//
if (((UINT32 *)&((EFI_FFS_FILE_HEADER *)ptr)->Name))[0]
!= 0x912740BE) || (((UINT32 *)&((EFI_FFS_FILE_HEADER *)ptr)-
>Name))[1] != 0x47342284) || (((UINT32
*)&((EFI_FFS_FILE_HEADER *)ptr)->Name))[2] != 0xB08471B9)
|| (((UINT32 *)&((EFI_FFS_FILE_HEADER *)ptr)->Name))[3] !=
0x0C3F3527) {
    ptr = 0;
    goto NotFound;
}

//
// Add the FFS Header size to the base to find the Raw
section Header
//
ptr += sizeof(EFI_FFS_FILE_HEADER);
if (((EFI_RAW_SECTION *)ptr)->Type != EFI_SECTION_RAW) {
    ptr = 0;
    goto NotFound;
}

//
// Add the Raw Header size to the base to find the FSP INFO
Header
//
ptr += sizeof(EFI_RAW_SECTION);

NotFound:
    __asm__ __volatile__ ("ret");
}

```

Now, call this function using a temporary ROM stack containing the return address and bypass the Prolog or Epilog code of the C function like below.

```

lea    findFspHeaderStack, %esp
jmp    find_fsp_entry

findFspHeaderStack:
    .align 4
    .long findFspHeaderDone

findFspHeaderDone:

```

§ §

