Intel® Firmware Support Package for Intel® Xeon® E3-1125C v2, E3-1105C v2, Intel® Pentium® Processor B925C, and Intel® Core™ i3-3115C Processors for Communications Infrastructure with Intel® Communications Chipset 89xx Series Platform Controller Hub

Integration Guide

March 2014

Revision 1.1

Contents

Revision History

Date	Revision	Description
March 2014	1.1	First public release.
November 2013	1.0	Initial release.

§

1 Introduction

1.1 Purpose

The purpose of this document is to describe the steps required to integrate the FSP into a boot loader solution.

1.2 Intelligent Systems and Embedded Ecosystem Overview

Contrasting 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 Firmware Support Package (FSP) to satisfy the needs of design engineers.

1.3 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 developer, boot loader developer, system integrators, as well as end users.

1.4 Related Documents

- Platform Initialization (PI) Specification located at http://www.uefi.org/specifications/.
- Intel® Firmware Support Package: Introduction Guide – available at www.intel.com/fsp
- Binary Configuration Tool for Intel® FSP – CDI Doc #520839
- *Binary Configuration Tool User Guide* – CDI Doc #518569

1.5 Conventions

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

1.6 Acronyms and Terminology

BSP	Boot Strap Processor
BWG	BIOS Writer's Guide
CRB	Customer Reference Board
FSP	Firmware Support Package
FSP API	Firmware Support Package Interface
FWG	Firmware Writer's Guide
IVI	In Vehicle Infotainment
NBSP	Node BSP
RSM	Resume to OS from SMM
SBSP	System BSP
SMI	System Management Interrupt
SMM	System Management Mode
TSEG	Memory Reserved at the Top of Memory to be used as SMRAM

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 specifications, porting guides, and reference code.

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 will perform all the necessary initialization steps as documented in the BWG 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 initialization.

3 FSP Integration

The FSP binary can be integrated easily into many different boot loaders, such as Coreboot, etc. and also into embedded OS directly.

Below are some required steps for the integration:

- Customizing

  The static FSP configuration parameters are part of the FSP binary and can be customized by external tools that will be 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 address specified during building the FSP.

- 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

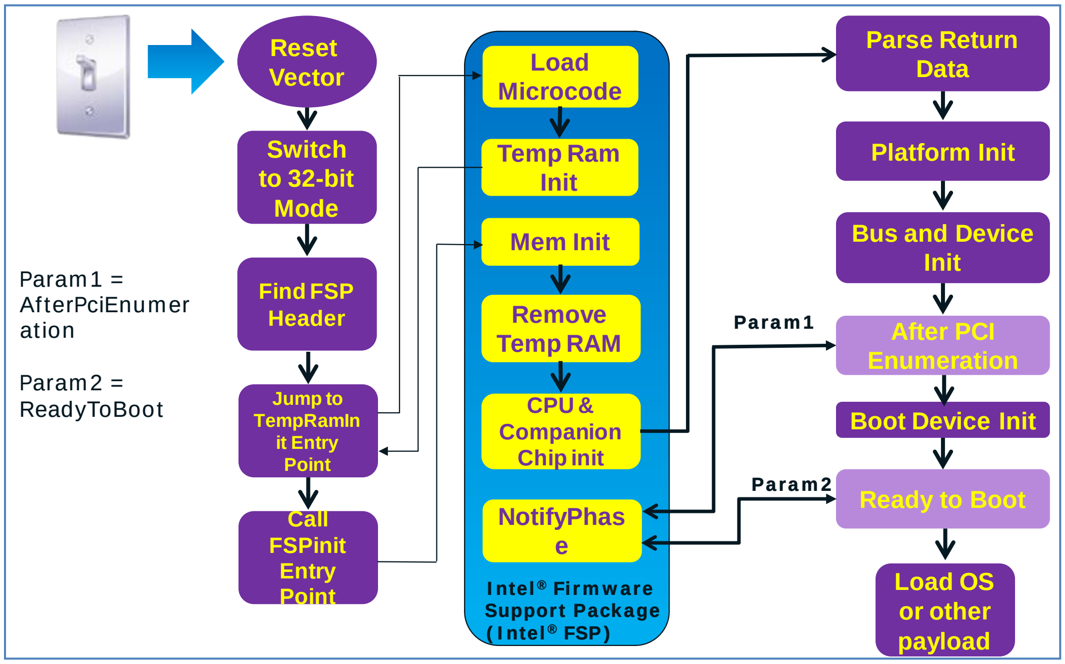
  The boot loader needs to add code to setup the operating environment for the FSP, call the FSP with the correct parameters and parse the FSP output to retrieve the necessary information returned by the FSP.

3.1 Assumptions Used in this Document

Because the Intel® Xeon® E3-1125C v2, E3-1105C v2, Intel® Pentium® Processor B925C, and Intel® Core™ i3-3115C Processors for Communications Infrastructure with Intel® Communications Chipset 89xx FSP is built with a preferred base address of 0xFFF80000, the FSP binary is assumed to be placed at the same address as part of the boot loader build.

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 a tool 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. For the Intel® Xeon® E3-1125C v2, E3-1105C v2, Intel® Pentium® Processor B925C, and Intel® Core™ i3-3115C Processors for Communications Infrastructure with Intel® Communications Chipset 89xx Platform Controller Hub, the default ImageBase is oxFFF80000.
32	4	ImageAttribute	Attributes of the FSP binary. This field is not currently used.

Byte Offset	Size in Bytes	Field	Description
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.
44	4	ApiEntryNum	Number of API Entries this 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 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 and 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.
- Please refer to Appendix – B for a sample code snippet which does the above steps in a stackless environment.

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 Intel® Xeon® E3-1125C v2, E3-1105C v2, Intel® Pentium® Processor B925C, and Intel® Core™ i3-3115C Processors for Communications Infrastructure with Intel® Communications Chipset 89xx Platform Controller Hub FSP, the FSP header is placed at an offset of **0x94.** So, for example, if the FSP binary is placed at **0xFFF80000** after the final build, the FSP header can be located at **0xFFF80094**. This implies that

1. The offset of the TempRamInitEntry can be found at **0xFFF800C4**
2. The offset of the FspInitEntry can be found at **0xFFF800C8**
3. The offset of the NotifyPhaseEntry can be found at **0xFFF800CC**

6 FSP Interface (FSP API)

6.1 Entry-Point Calling Assumptions

There are some requirements regarding the operating environment for FSP execution. It is the responsibility of the boot loader 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:

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

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

6.2 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 a 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. Please refer to individual API descriptions for any special notes and these exceptions.

6.3 Exit Convention

- The TempRamInit API will preserve 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/return 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 will be passed back though a HOB. 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.

6.4 TempRamInitEntry

This FSP API is called soon after coming out of reset and before memory and stack are 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 till main memory is initialized.

A hardcoded stack can be setup 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 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 of 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.

6.4.1 Prototype

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

6.4.2 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 are compatible with the silicon it is supporting.

This microcode region will be remembered by FSP so that it can be used to load microcode for all APs later on during the FspInit API call.

6.4.3 Related Definitions

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

6.4.3.1 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 processor. FSP maintains and consumes the remaining temporary memory. It is important for the boot loader not to access the temporary memory beyond the returned boundary.

FSP_SUCCESS	Temp RAM was initialized successfully. ESP register will be initialized as described below.
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.

6.4.3.2 Sample Code

```
.global basic_init
basic_init:
  .
  .
  .

  #
  # Parse the FV to find the FSP INFO Header
  #
  lea   findFspHeaderStack, %esp
  jmp   find_fsp_info_header
findFspHeaderDone:
  mov   %eax,   %ebp               # save fsp header address in
ebp
  mov   0x30(%ebp), %eax           # TempRamInit offset entry in
the header
  add   0x1c(%ebp), %eax           # add the FSP base to get the
API address

  lea   tempRamInitStack, %esp      # initialize to a rom stack

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

temp_RamInit_done:
  addl  $4, %esp

   cmp   $0, %eax
  jz    continue
```

```
    #
    # TempRamInit failed, dead loop
    #
    jmp    .

  continue:
    #
    # Save FSP_INFO_HEADER in ebx
    #
    mov    %ebp, %ebx

    #
    # setup bootloader stack
    # ecx:  stack base
    # edx:  stack top
    #
    lea    -4(%edx), %esp

    #
    # call C based function to initialize meomry and chipset. Pass
    # the FSP INFO Header address as a parameter
    #
    push   %ebx
    call   early_init


    #
    # should never return here
    #
    jmp   .

    .align 4
  findFspHeaderStack:
    .long  findFspHeaderDone

  tempRamInitParams:
    .long  _ucode_base      # Microcode base address
    .long  _ucode_size      # Microcode size
    .long   0xfff80000      # Code Region Base
    .long   0x00040000      # Code Region Length

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

Note: early_init(FSP_INFO_HEADER *fsp_info) is described in section 6.5.5.

6.4.4 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 calls 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 initializes 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 EDX and ECX.

6.5 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 with each FSP release.

The boot loader provides a continuation function as a parameter when calling FspInit. After FspInit completes its execution, it will not return to the boot loader from where it was called, but instead will return control to the boot loader by calling the continuation function.

6.5.1 Prototype

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

6.5.2 Parameters

FspInitParamPtr Address pointer to the FSP_INIT_PARAMS structure

6.5.3 Related Definitions

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

NvsBufferPtr	Pointer to the non-volatile storage data buffer.
RtBufferPtr	Pointer to the runtime data buffer.
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.

The FSP_INIT_RT_BUFFER for the Intel® Xeon® E3-1125C v2, E3-1105C v2, Intel® Pentium® Processor B925C, and Intel® Core™ i3-3115C Processors for Communications Infrastructure with Intel® Communications Chipset 89xx Platform Controller Hub FSP is defined as below.

```
typedef struct {
  UINT32              *StackTop;
  UINT32              BootMode;

  VOID               *UpdDataRegPtr;
  UINT32             Reserved[7];
} FSP_INIT_RT_COMMON_BUFFER;

typedef struct {
  const MEM_CONFIG     *MemoryConfig;
} FSP_INIT_RT_PLATFORM_BUFFER;

typedef struct {
  FSP_INIT_RT_COMMON_BUFFER   Common;
  FSP_INIT_RT_PLATFORM_BUFFER RtPlatform;
} FSP_INIT_RT_BUFFER;
```

MEM_CONFIG is the data structure for Memory Down. Refer Appendix C for its structure definitions.

6.5.4 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 meet.
FSP_DEVICE_ERROR	FSP initialization failed

6.5.5 Sample Code

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

typedef struct {

  void                *NvsBufferPtr;

  void                *RtBufferPtr;

  CONTINUATION_PROC   ContinuationFunc;

} FSP_INIT_PARAMS;

#define FSPAPI __attribute__((cdecl))

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

#define BOOTLOADER_STACKTOP 0x98000 // Porting required

 void early_init (FSP_INFORMATION_HEADER *fsp_info)

{

    .

    .

    .


  uint32_t                           FspInitEntry;

  FSP_FSP_INIT                       FspInitApi;

  volatile FSP_INIT_PARAMS           FspInitParams;

  volatile FSP_INIT_RT_BUFFER        FspRtBuffer;
```

```
UPD_DATA_REGION   FspUpdRgn;

VPD_DATA_REGION   *FspVpdRgn;

FSP_INIT_PARAMS   *FspInitParamsPtr;

uint32_t          UpdDataOffInStack;


  memset((void*)&FspRtBuffer, 0, sizeof(FSP_INIT_RT_BUFFER));

  FspRtBuffer.Common.StackTop = BOOTLOADER_STACKTOP;

  FspRtBuffer.Common.UpdDataRgnPtr = (UPD_DATA_REGION
*)&FspUpdRgn;

  FspVpdRgn = (VPD_DATA_REGION *)(fsp_info->ImageBase + fsp_info-
>CfgRegionOffset);

  memcpy (&FspUpdRgn, (void *)(fsp_infp->ImageBase + FspVpdRgn-
>PcdUpdRegionOffset), sizeof(UPD_DATA_REGION));

  FspRtBuffer.RtPlatform.MemoryConfig =NULL;

  FspInitParams.NvsBufferPtr = 0;

  FspInitParams.RtBufferPtr = (FSP_INIT_RT_BUFFER *)&FspRtBuffer;

  FspInitParams.ContinuationFunc =
    (CONTINUATION_PROC)ContinuationFunc;

  FspInitApi = (FSP_FSP_INIT)(fsp_info->ImageBase + fsp_info
    ->FspInitEntry);


FspInitParamPtr = &FspInitParams;

UpdDataOffStack = (uint_32 *)&FspUpdRgn - (uint_32
*)(stack_base);


asm volatile(

         "push %0;"

         "call *%%eax;"

         : : "m"(FspInitParamPtr), "a"(FspInitApi),
"b"(fsp_info), "c"(UpdDataOffInStack));

/*
```

```
     * Should never get here. Control will continue from
     * romstage_main_continue_asm

     * This line below is to prevent the compiler from optimizing
     * structure intialization

     */


    FspInitApi(&FspInitParams);

    while (1);

  }


  void ContinuationFunc (EFI_STATUS Status, VOID *HobListPtr)

  {

   /* Check status for -1, indicating that we need to do a reset */

      if (Status == 0xFFFFFFFF)

      {

          /* Trigger the reset */

          outb(0x0cf9, 0x06);


          /* Should never return */

          while (1);

      }


      /* Update global variables */

      FspHobListPtr = HobListPtr;


  /* Get the pointer to the FSP header the same way as in
   basic_init */

  asm ("call find_fsp_info_header; movl %%eax, %0" : "=r"
  (fsp_info_header) :);
```

```
    /* Continue the boot */
    advancedInit ();


    /* Should never return */
    while (1);
}
```

6.5.6 Enabling Fast Boot

The shaded lines in the sample code below highlight the things that need to be changed to enable fast boot in the bootloader. First, the FastBootEnable field needs to be set to "1" in the UPD region created in the sample code above.

```
FspUpdRgn.FastBootEnable = 1;
```

The boot loader should implement code to save and read back the HOB data to/from non-volatile storage. The sample code below shows an example implementation for passing the HOB data back to the FSP thru the NvsBufferPtr pointer. For the first boot, the data buffer that this pointer points to should contain either all zeroes or all 0xFFs so that the FSP can recognize it as being uninitialized. For subsequent boots, the data buffer that this pointer points to should contain valid memory training data that was generated by the FSP during the first boot and was saved by boot loader to the data buffer in non-volatile storage.

```
void early_init (FSP_INFO_HEADER *fsp_info)
{
  FSP_FSP_INIT          FspInitEntry;
  FSP_INIT_PARAMS       FspInitParams;
  FSP_INIT_RT_BUFFER    FspRtBuffer;
  VOID                 *NvsBufferPtr;

  ...  // Some init code.

  // Get pointer to saved NVSTORAGE HOB data
  NvsBufferPtr = ...  // Platform-specific

  memset((void*)&FspRtBuffer, 0, sizeof(FSP_INIT_RT_BUFFER));
  FspRtBuffer.Common.StackTop = &_stack_top;
  FspRtBuffer.PlatformConfiguration.PlatformConfig =
&PlatformConfig;
  FspRtBuffer.Platform.MemoryConfig = NULL;
  FspInitParams.NvsBufferPtr = NvsBufferPtr;
  FspInitParams.RtBufferPtr = (FSP_INIT_RT_BUFFER *)&FspRtBuffer;
  FspInitParams.ContinuationFunc =
(CONTINUATION_PROC)ContinuationFunc;
  FspInitEntry = (FSP_FSP_INIT)(fsp_info->ImageBase + fsp_info-
>FspInitEntry);
  FspInitEntry(&FspInitParams);

  /* Should never return. Control will continue from
ContinuationFunc */
  while (1);
}

After FspInitEntry is done, control is returned to the boot
loader via the ContinuationFunc function, which is passed a
pointer to the HOB list. The boot loader should save this pointer
for use later.

volatile void   *FspHobListPtr;

void ContinuationFunc (EFI_STATUS Status, VOID *HobListPtr)
{
    ...

    /* Update global variables */
    FspHobListPtr = HobListPtr;

    ...

    /* Continue the boot */
    advancedInit ();

    /* Should never return */
    while (1);
}
```

One of the HOBs in the HOB list contains the Non-Volatile Storage data, which includes the memory training data required for fast boot. It's the data from this HOB that must be saved into non-volatile storage to be used in subsequent boots. This data is extracted from the HOB list with the following function, which depends upon values and functions that are defined in fsphob.h and are implemented fsphob.c, both of which are included with the files that are packaged with the FSP.

```
#include "fsphob.h"

void GetFspHobDataForNVStorage(
  VOID         **ppHobData,
  uint16_t     *pHobDataSize
  )
{
  EFI_GUID                Guid =
FSP_NON_VOLATILE_STORAGE_HOB_GUID;
  uint8_t                 *Hob;
  EFI_HOB_GENERIC_HEADER  *HobHdr;

  Hob = GetFirstGuidHob(&Guid);

  if (!Hob) {
    *ppHobData = NULL;
    *pHobDataSize = 0;
  } else {
    *ppHobData = (VOID *)GET_GUID_HOB_DATA(Hob);
    HobHdr = (EFI_HOB_GENERIC_HEADER *)Hob;
    *pHobDataSize =  GET_GUID_HOB_DATA_SIZE(HobHdr);
  }
}
```

This function is called from the boot loader after everything else is successfully initialized, right before calling FspNotifyPhase(EnumInitPhaseReadyToBoot), and the HOB data is saved to non-volatile storage. Sample code below shows that this function is called inside advancedInit.

```
void advancedInit()
{
    VOID *HobData;
    uint16_t HobDataSize;

    ...

    /* NVS data to save */
    GetFspHobDataForNVStorage(&HobData, &HobDataSize);

    /* Check result */
    if ((HobData != NULL) && (HobDataSize > 0))
    {
        /* Check if data will fit in available NV storage */
        if (HobDataSize <= NVRAM_SIZE) // Platform-specific
        {
            /* Save NVSTORAGE HOB data
            ...  // Platform-specific
        }
    }

    /* Notify FSP for ReadyToBoot */
    FspNotifyPhase(EnumInitPhaseReadyToBoot);

    ...

}
```

6.5.7 Description

One of the data that will be part of the FSP_INIT_PARAMS. RtBufferPtr will be the "StackTop". This will pass the address of the StackTop where the boot loader wants to establish the stack once memory is initialized and available for use.
ContinuationFunc is a function entry point that will be jumped to at the end of the FspInit() execution to transfer control back to the boot loader.
Please note the FspInit API will initialize the permanent memory and switch the stack from temporary to permanent memory as specified by StackTop. Sometimes switching 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 will then require assembly code to patch up the data for the new stack location, which may lead to compatibility issues. To avoid such possible compatibility issues introduced by different compilers and ease the integration of FSP with a boot loader, the API will use the "ContinuationFunction" parameter to continue the boot loader execution flow rather than return as a normal C function. Although this API will be called as a normal C function, it will never return to one.

The FSP will need to get some parameters from the boot loader when it's initializing the silicon. These parameters are passed from the boot loader to the FSP through the RtBuffer structure pointer.

The FSP may need to initialize memory under special circumstances, such as during an S3 resume and fast boot mode. This set of parameters will be returned by the FSP to the boot loader during a normal boot. The boot loader is expected to store these parameters in a non-volatile memory, comparable to 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.

During execution the FSP will build a series of data structures containing information useful to the boot loader, such as information on system memory.

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

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 FSP across different platforms

It's left to the boot loader developer to decide on how to consume the information passed through the HOBs produced by the FSP. For example, even the specification mentioned above describes about 9 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 of 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.

Please refer to the specification for details about the HOBs described in the *Platform Initialization (PI) specification - Volume 3: Shared Architectural Elements* specification.

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 boot loader even after the FspInit call. So before destroying the temporary memory, all contents in this sub region is migrated to the permanent memory. FSP builds a boot loader temporary memory data HOB, which the boot loader can 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, 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 } };
```

7.2 Non-Volatile Storage HOB

The NVS buffer that is used by FSP to initialize the silicon during S3 resume, fast boot mode, etc., is passed through a GUID HOB with a GUID HOB defined as below. The boot loader is expected to save the data in a non-volatile storage memory area and initialize the FspInit parameter properly when requesting special initialization sequences such as S3 resume, fast boot mode, etc.

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

```
#define FSP_NON_VOLATILE_STORAGE_HOB_GUID \

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

7.3 HOB Sample Code

An example function using the HOB infrastructure and getting the memory information is provided below.

7.3.1 Hob Infrastructure Sample Code

Please refer to the Appendix - A for sample code.

7.3.2 Hob Parsing Sample Code

```
void
GetMemorySize (
  UINT32           *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) (Hob.ResourceDescriptor-
>ResourceLength);
        }
      }
    }
    Hob.Raw = GET_NEXT_HOB (Hob);
  }

  return;
}


 void boot_loader_rom_stage_fn ()
{

    .
    .
    .


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

  /* Get the memory size by parsing the HOB returned from the FSP
*/
  GetMemorySize (&LowMemoryLength, FspInitParams.HobBufferPtr);
    .
    .
    .
}
```

7.3.3 GUID HOB Sample Code

```
void *
GetGuidHobData (
  CONST EFI_GUID         *Guid
  )
{
  VOID  *GuidHob;

  GuidHob = GetFirstGuidHob (Guid);
  if (GuidHob == NULL) {
    return NULL;
  }
  return (void *)GET_GUID_HOB_DATA (GuidHob);
}
```

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 be configured statically

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

Both the VPD and the UPD parameters can be statically customized using a separate tool called the Binary Configuration Tool as explained in the tools section. The tool will use a Binary 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 a 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 UPD data that is available in the FSP configuration region. However, if the boot loader wishes to override any of the UPD parameters, it has to copy the UPD structure to memory, override the parameters and initialize the pointer in the FspInit API input parameter to the address of the UPD structure with updated data in memory and call FspInit API. The FSP will use this data structure instead of the default configuration region data.

When calling the FspInit API, the stack is in temporary RAM where the UPD data structure is copied, updated, and passed to the FSP API. When permanent memory is initialized, the FSP will set up a new stack in the permanent memory. However, the FSP will save the stack that was in the Temporary Memory in a HOB. If the boot loader wishes to refer to the modified UPD Data, it can be done by parsing the HOB which has the Temporary Stack's data.

Both the VPD and the UPD structure definition is provided below. As mentioned above, to update these configuration regions statically using the Binary Configuration Tool, a BSF file will be provided separately.

```
typedef struct _UPD_DATA_REGION {
  UINT64   Signature;                      /* Offset 0x0000 */
  UINT64   Reserved;                       /* Offset 0x0008 */
  UINT8    HTEnable;                       /* Offset 0x0010 */
  UINT8    TurboEnable;                    /* Offset 0x0011 */
  UINT8    MemoryDownEnable;               /* Offset 0x0012 */
  UINT8    FastBootEnable;                 /* Offset 0x0013 */
  UINT8    ReservedUpdSpace0[229];         /* Offset 0x0014 */
  UINT16   PcdRegionTerminator;            /* Offset 0x00F9 */
} UPD_DATA_REGION;
```

```
typedef struct _VPD_DATA_REGION {
  UINT64    PcdVpdRegionSign;                   /* Offset 0x0000 */
  UINT32    PcdImageRevision;                   /* Offset 0x0008 */
  UINT32    PcdUpdRegionOffset;                 /* Offset 0x000C */
  UINT32    PcdFspReservedMemoryLength;         /* Offset 0x0010 */
  UINT64    Dummy1;                             /* Offset 0x0014 */
  UINT32    Dummy2;                             /* Offset 0x001C */
  UINT32    PcdBootLoaderEntry;                 /* Offset 0x0020 */
  UINT32    PcdMeStolenSize;                    /* Offset 0x0024 */
  UINT8     PcdVgaMemoryEnable                  /* Offset 0x0028 */
  UINT8     PcdFastTrainingMode;                /* Offset 0x0029 */
  UINT32    PcdTsegSize;                        /* Offset 0x002A */
  UINT64    PcdSpdAddress;                      /* Offset 0x002E */
  UINT8     PcdEccSupport;                      /* Offset 0x0036 */
  UINT16    PcdDdrFreqLimit;                    /* Offset 0x0037 */
  UINT8     PcdNModeSupport;                    /* Offset 0x0039 */
  UINT8     PcdScramblerSupport;                /* Offset 0x003A */
  UINT8     PcdRmtCrosserEnable;                /* Offset 0x003B */
  UINT8     PcdThermalManagement;               /* Offset 0x003C */
  UINT8     PcdPowerDownMode;                   /* Offset 0x003D */
  UINT8     PcdDisableDimmChannel0;             /* Offset 0x003E */
  UINT8     PcdDisableDimmChannel1;             /* Offset 0x003F */
  UINT8     PcdDDR3Voltage;                     /* Offset 0x0040 */
  UINT8     PcdDmiVc1;                          /* Offset 0x0041 */
  UINT8     PcdDmiVcp;                          /* Offset 0x0042 */
  UINT8     PcdDmiVcm;                          /* Offset 0x0043 */
  UINT8     PcdDmiGen2;                         /* Offset 0x0044 */
  UINT32    PcdPegGenx;                         /* Offset 0x0045 */
  UINT16    PcdMmioSize;                        /* Offset 0x0049 */
  UINT16    PcdGttSize;                         /* Offset 0x004B */
  UINT8     PcdIgdDvmt50PreAlloc;               /* Offset 0x004D */
  UINT8     PcdAlwaysEnablePeg;                 /* Offset 0x004E */
  UINT8     PcdInternalGraphics;                /* Offset 0x004F */
  UINT8     PcdPrimaryDisplay;                  /* Offset 0x0050 */
  UINT8     PcdApertureSize;                    /* Offset 0x0051 */
  UINT8     PcdHpetEnable;                      /* Offset 0x0052 */
  UINT8     PcdDmiAspm;                         /* Offset 0x0053 */
  UINT8     PcdPegAspm;                         /* Offset 0x0054 */
  UINT8     PcdDmiExtSync;                      /* Offset 0x0055 */
  UINT8     PcdDeEmphasis;                      /* Offset 0x0056 */
  UINT8     PcdFeatureConfigure;                /* Offset 0x0057 */
  UINT8     PcdLimitCpuidMaximumValue;          /* Offset 0x0058 */
  UINT8     PcdVmxEnable;                       /* Offset 0x0059 */
  UINT8     PcdTxtEnable;                       /* Offset 0x005A */
  UINT8     PcdMonitorMwaitEnable;              /* Offset 0x005B */
  UINT8     PcdExecuteDisableBit;               /* Offset 0x005C */
  UINT8     PcdFastString;                      /* Offset 0x005D */
  UINT8     PcdMachineCheckEnable;              /* Offset 0x005E */
  UINT8     PcdXapicEnable;                     /* Offset 0x005F */
  UINT8     PcdPchSata;                         /* Offset 0x0060 */
  UINT8     PcdPchSmbus;                        /* Offset 0x0061 */
```

```
    UINT8     PcdPchPciClockRun;                        /* Offset 0x0062 */
    UINT8     PcdPchEhci;                               /* Offset 0x0063 */
    UINT8     PcdDcaEnable;                             /* Offset 0x0064 */
    UINT64    PcdPchPciePortEn;                         /* Offset 0x0065 */
    UINT64    PcdPchPciePortSIE;                        /* Offset 0x006D */
    UINT64    PcdPchPciePortAspm;                       /* Offset 0x0075 */
    UINT64    PcdPchPciePortExtSync;                    /* Offset 0x007D */
    UINT64    PcdPchPciePortURE;                        /* Offset 0x0085 */
    UINT64    PcdPchPciePortFEE;                        /* Offset 0x008D */
    UINT64    PcdPchPciePortNFE;                        /* Offset 0x0095 */
    UINT64    PcdPchPciePortCEE;                        /* Offset 0x009D */
    UINT64    PcdPchPciePortCTD;                        /* Offset 0x00A5 */
    UINT64    PcdPchPciePortPIE;                        /* Offset 0x00AD */
    UINT64    PcdPchPciePortSFE;                        /* Offset 0x00B5 */
    UINT64    PcdPchPciePortSNE;                        /* Offset 0x00BD */
    UINT64    PcdPchPciePortSCE;                        /* Offset 0x00C5 */
    UINT64    PcdPchPciePortPMCE;                       /* Offset 0x00CD */
    UINT64    PcdPchPciePortHPCE;                       /* Offset 0x00D5 */
    UINT8     PcdPchSataMode;                           /* Offset 0x00DD */
    UINT64    PcdPchSataPortEn;                         /* Offset 0x00DE */
    UINT64    PcdPchSataPortHP;                         /* Offset 0x00E6 */
    UINT64    PcdPchSataPortIS;                         /* Offset 0x00EE */
    UINT64    PcdPchSataPortEX;                         /* Offset 0x00F6 */
    UINT64    PcdPchSataPortMUL;                        /* Offset 0x00FE */
    UINT8     PcdPchWatchDog;                           /* Offset 0x0106 */
    UINT8     PcdPchUsbPerPortCtl;                      /* Offset 0x0107 */
    UINT8     PcdPchDmiAspm;                            /* Offset 0x0108 */
  } VPD_DATA_REGION;
```

9 Tools

A Binary Configuration Tool (BCT) will be provided with the FSP binary that can used on the FSP binary to allow a user to modify certain well defined configuration values in the FSP binary. The BCT will typically provide a graphical user interface (GUI). The Binary Configuration Tool (BCT) will be provided with separate documentation that explains the usage of the tool.

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 will not be included in Intel® FSP.

10.2 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.

10.3 Security

Intel® FSP does not provide security features besides making security features available to the host boot loader.

10.4 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.

10.5 Pre-OS Graphics

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

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                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 {
  ///
  /// 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 Serves 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 2.0specification.
  ///
  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
  /// memory 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
  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  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 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 alignment 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 FSP Header

The sample code provided below parses the FSP binary and finds the address of the 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 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.

The sample code provided here uses header files derived from the EDK2 source available for download at

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

```
#include "PiFirmwareVolume.h"
#include "PiFirmwareFile.h"

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

    __asm__ __volatile__ (
        ".global find_fsp_info_header \n\t"
        "find_fsp_info_header:\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)->ExtHeaderOffset;
    ptr += ((EFI_FIRMWARE_VOLUME_EXT_HEADER *)ptr)-
>ExtHeaderSize;
```

```
    //
    // Align the pointer to 8 bytes and it will point to FFS
    // header
    //
    ptr  = (UINT8 *)(((UINTN)ptr + 7) & 0xFFFFFFF8);

    //
    // Now ptr is pointing to thr 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");

}
```

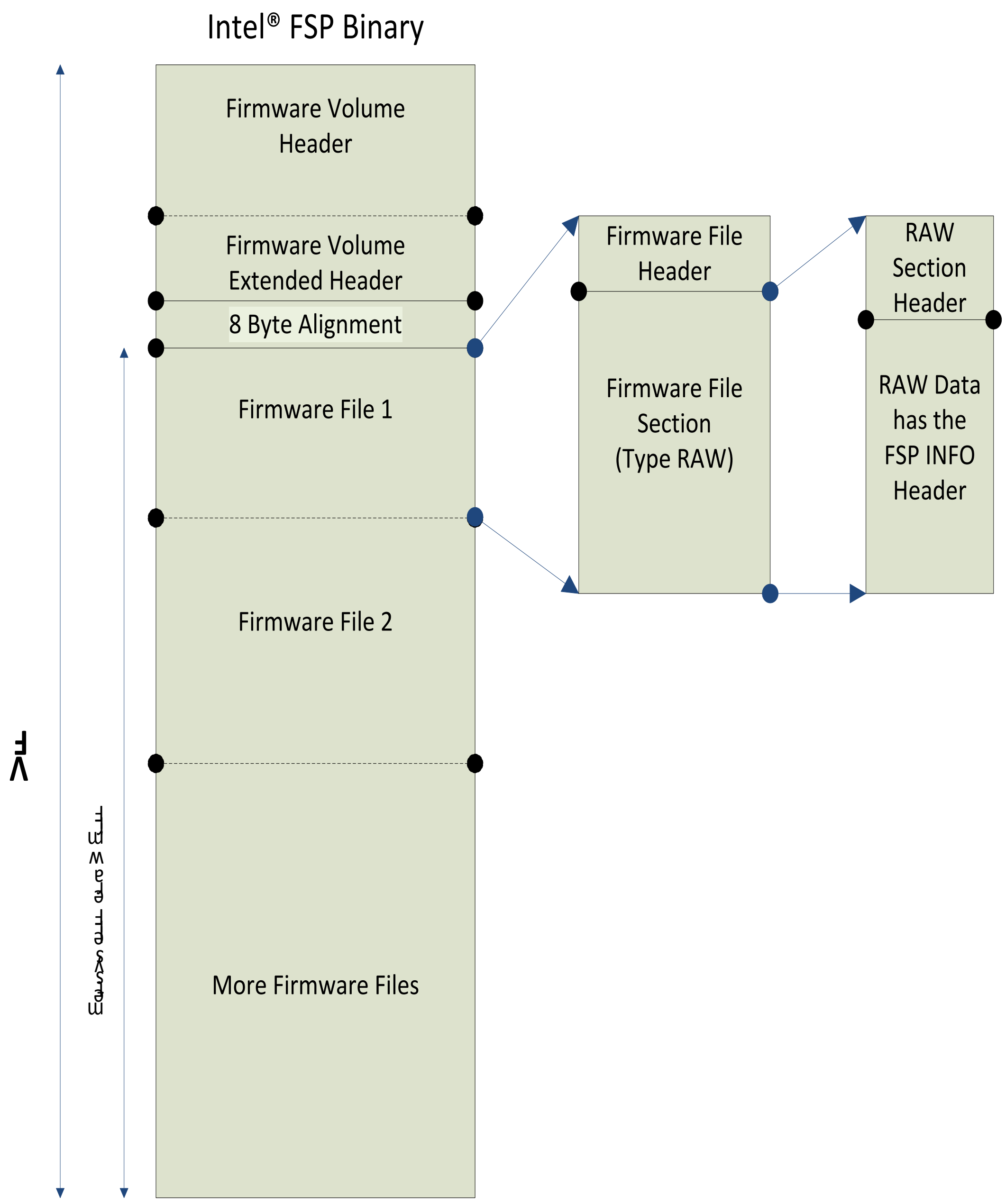
Now, call this function using a temporary ROM stack containing the return address and bypass the prolog or epilog code of the C function like below.

```
  lea   findFspHeaderStack, %esp
  jmp   find_fsp_entry

 findFspHeaderStack:
  .align 4
  .long  findFspHeaderDone

 findFspHeaderDone:
```

A pictorial representation of the data structures that we parse in the above code is given below.



Appendix C – Data Structure for Memory Down

The data structures provided below are defined for Memory Down and need to be passed to FSP when the Memory Down feature is enabled:

```
#define NUM_IVB_MEM_CLK_FREQUENCIES 13


typedef struct {
  // 0   Number of Serial PD Bytes Written / SPD Device Size /
     CRC Coverage 1, 2
  u8  SPDGeneral;
  // 1   SPD Revision
  u8  SPDRevision;
  // 2   DRAM Device Type
  u8  DRAMDeviceType;
  // 3   Module Type
  u8  ModuleType;
  // 4   SDRAM Density and Banks
  u8  SDRAMDensityAndBanks;
  // 5   SDRAM Addressing
  u8  SDRAMAddressing;
  // 6   Module Nominal Voltage
  u8  VDD;
  // 7   Module Organization
  u8  ModuleOrganization;
  // 8   Module Memory Bus Width
  u8  ModuleMemoryBusWidth;
  // 9   Fine Timebase (FTB) Dividend / Divisor
  u8  FineTimebase;
  // 10  Medium Timebase (MTB) Dividend
  u8  TimebaseDividend;
  // 11  Medium Timebase (MTB) Divisor
  u8  TimebaseDivisor;
  // 12  SDRAM Minimum Cycle Time (tCKmin)
  u8  SDRAMMinimumCycleTime;
  // 13  Reserved0
  u8  Reserved0;
  // 14  CAS Latencies Supported, Least Significant Byte
  u8  CASLatenciesLSB;
  // 15  CAS Latencies Supported, Most Significant Byte
  u8  CASLatenciesMSB;
  // 16  Minimum CAS Latency Time (tAAmin)
  u8  MinimumCASLatencyTime;
  // 17  Minimum Write Recovery Time (tWRmin)
  u8  MinimumWriteRecoveryTime;
```

```
// 18  Minimum RAS# to CAS# Delay Time (tRCDmin)
u8  MinimumRASToCASDelayTime;
// 19  Minimum Row Active to Row Active Delay Time (tRRDmin)
u8  MinimumRowToRowDelayTime;
// 20  Minimum Row Precharge Delay Time (tRPmin)
u8  MinimumRowPrechargeDelayTime;
// 21  Upper Nibbles for tRAS and tRC
u8  UpperNibblesFortRASAndtRC;
// 22  Minimum Active to Precharge Delay Time (tRASmin), Least
   Significant Byte
u8  tRASmin;
// 23  Minimum Active to Active/Refresh Delay Time (tRCmin),
   Least Significant Byte
u8  tRCmin;
// 24  Minimum Refresh Recovery Delay Time (tRFCmin), Least
 Significant Byte
u8  tRFCminLeastSignificantByte;
// 25  Minimum Refresh Recovery Delay Time (tRFCmin), Most
 Significant Byte
u8  tRFCminMostSignificantByte;
// 26  Minimum Internal Write to Read Command Delay Time
 (tWTRmin)
u8  tWTRmin;
// 27  Minimum Internal Read to Precharge Command Delay Time
 (tRTPmin)
u8  tRTPmin;
// 28  Upper Nibble for tFAW
u8  UpperNibbleFortFAW;
// 29  Minimum Four Activate Window Delay Time (tFAWmin)
u8  tFAWmin;
// 30  SDRAM Optional Features
u8  SDRAMOptionalFeatures;
// 31  SDRAMThermalAndRefreshOptions
u8  SDRAMThermalAndRefreshOptions;
// 32  ModuleThermalSensor
u8  ModuleThermalSensor;
// 33  SDRAM Device Type
u8  SDRAMDeviceType;
// 34  Fine Offset for SDRAM Minimum Cycle Time (tCKmin)
s8  tCKminFine;
// 35  Fine Offset for Minimum CAS Latency Time (tAAmin)
s8  tAAminFine;
// 36  Fine Offset for Minimum RAS# to CAS# Delay Time
 (tRCDmin)
s8  tRCDminFine;
// 37  Fine Offset for Minimum Row Precharge Delay Time
 (tRPmin)
s8  tRPminFine;
// 38  Fine Offset for Minimum Active to Active/Refresh Delay
 Time (tRCmin)
s8  tRCminFine;
// 62  Reference Raw Card Used
u8  ReferenceRawCardUsed;
```

```
  // 63  Address Mapping from Edge Connector to DRAM
  u8  AddressMappingEdgeConnector;
  // 64  ThermalHeatSpreaderSolution
  u8  ThermalHeatSpreaderSolution;
  // 117 Module Manufacturer ID Code, Least Significant Byte
  u8  ModuleManufacturerIdCodeLsb;
  // 118 Module Manufacturer ID Code, Most Significant Byte
  u8  ModuleManufacturerIdCodeMsb;
  // 119 Module Manufacturing Location
  u8  ModuleManufacturingLocation;
  // 120 Module Manufacturing Date Year
  u8  ModuleManufacturingDateYear;
  // 121 Module Manufacturing Date creation work week
  u8  ModuleManufacturingDateWW;
  // 122 Module Serial Number A
  u8  ModuleSerialNumberA;
  // 123 Module Serial Number B
  u8  ModuleSerialNumberB;
  // 124 Module Serial Number C
  u8  ModuleSerialNumberC;
  // 125 Module Serial Number D
  u8  ModuleSerialNumberD;
  // 126 CRC A
  u8  CRCA;
  // 127 CRC B
  u8  CRCB;
} DDR3_SPD;


typedef struct {
  u32       Exists;
  DDR3_SPD  SpdData;
  u8        InitClkPiValue[NUM_IVB_MEM_CLK_FREQUENCIES];
} MEM_BANK_CONFIG;


typedef struct {
  MEM_BANK_CONFIG  ChannelABank0;
  MEM_BANK_CONFIG  ChannelABank1;
  MEM_BANK_CONFIG  ChannelBBank0;
  MEM_BANK_CONFIG  ChannelBBank1;
} MEM_CONFIG;
```

C.1 Sample Code

Memory Down support is required when the hardware design does not use standard DIMMs or SO-DIMMs, but instead has the memory soldered down to the board, and the serial presence detect (SPD) data is not stored in a standard SPD EEPROM such as is found on a standard DIMM or SO-DIMM. In this situation, the boot loader must provide a means to pass the SPD data to the FSP.

The shaded lines in the sample code below highlight the things that need to be changed to support memory down.

To enable memory down, the `MemoryDownEnable` field needs to be set to TRUE in the PLATFORM_CONFIG structure:

```
// Platform Configuration
const PLATFORM_CONFIG PlatformConfig = {
  TRUE,   // Hyperthreading
  FALSE,  // Turbo Mode
  TRUE,  // Memory Down
  FALSE,   // Fast Boot
};
```

The boot loader must provide a board-specific means of reading the SPD data from where it is stored on the hardware using a function with a prototype similar to this:

```
void ReadSpdData(unsigned char Channel,
                 unsigned char Bank,
                 DDR3_DATA* SpdData);
```

The following sample code shows how to provide the SPD data to the FSP for a hardware design that implements memory on Channel A Bank 0 and Channel B Bank 0.

```
void early_init (FSP_INFO_HEADER *fsp_info)
{
  FSP_FSP_INIT              FspInitEntry;
  FSP_INIT_PARAMS           FspInitParams;
  FSP_INIT_RT_BUFFER        FspRtBuffer;
  MEM_CONFIG                MemoryConfig;

  ...  // Some init code.


// Initialize the memory config
 memset(&MemoryConfig, 0, sizeof(MemoryConfig));

  // Channel A Bank 0 and Channel B Bank 0 exist
  MemoryConfig.ChannelABank0.Exists = 1;
  MemoryConfig.ChannelBBank0.Exists = 1;
  // Read the SPD data from the hardware
  ReadSpdData(0, 0, &MemoryConfig.ChannelABank0.SpdData);  //
Platform-specific
  ReadSpdData(1, 0, &MemoryConfig.ChannelBBank0.SpdData);  //
Platform-specific


  memset((void*)&FspRtBuffer, 0, sizeof(FSP_INIT_RT_BUFFER));
  FspRtBuffer.Common.StackTop = &_stack_top;
  FspRtBuffer.PlatformConfiguration.PlatformConfig =
&PlatformConfig;
  FspRtBuffer.Platform.MemoryConfig = &MemoryConfig;
   FspInitParams.RtBufferPtr = (FSP_INIT_RT_BUFFER
*)&FspRtBuffer;
  FspInitParams.ContinuationFunc =
(CONTINUATION_PROC)ContinuationFunc;
  FspInitEntry = (FSP_FSP_INIT)(fsp_info->ImageBase + fsp_info-
>FspInitEntry);
  FspInitEntry(&FspInitParams);

  /* Should never return. Control will continue from
ContinuationFunc */
  while (1);
}
```

§