Intel® Firmware Support Package for the Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure

Integration Guide

April 2015

Contents

Revision History

Date	Revision	Description
April 2015	1.1	Updated Section 6.4 Exit Convention. Added Section 7.2 FSP Reserved Memory Resource Descriptor HOB. Added to Section 7.4.2 Hob Parsing Sample Code. Added to Section 8.1.2 UPD Data Region. Updated product name to Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure.
February 2014	1.0	Initial public release.

§

1 Introduction

1.1 Purpose

The purpose of this document is to describe the steps required to integrate the Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure Firmware Support Package (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/specs/.
- Intel® Firmware Support Package: Introduction Guide – available at http://www.intel.com/fsp
- Binary Configuration Tool for Intel® Firmware Support Package – available at www.intel.com/fsp
- Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure SoC SPI Flash Programming User Guide - CDI Doc #519715

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

Acronym	Description
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 performs 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 an 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 that is different from the preferred address specified during building the FSP.

- Placing

  After 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.

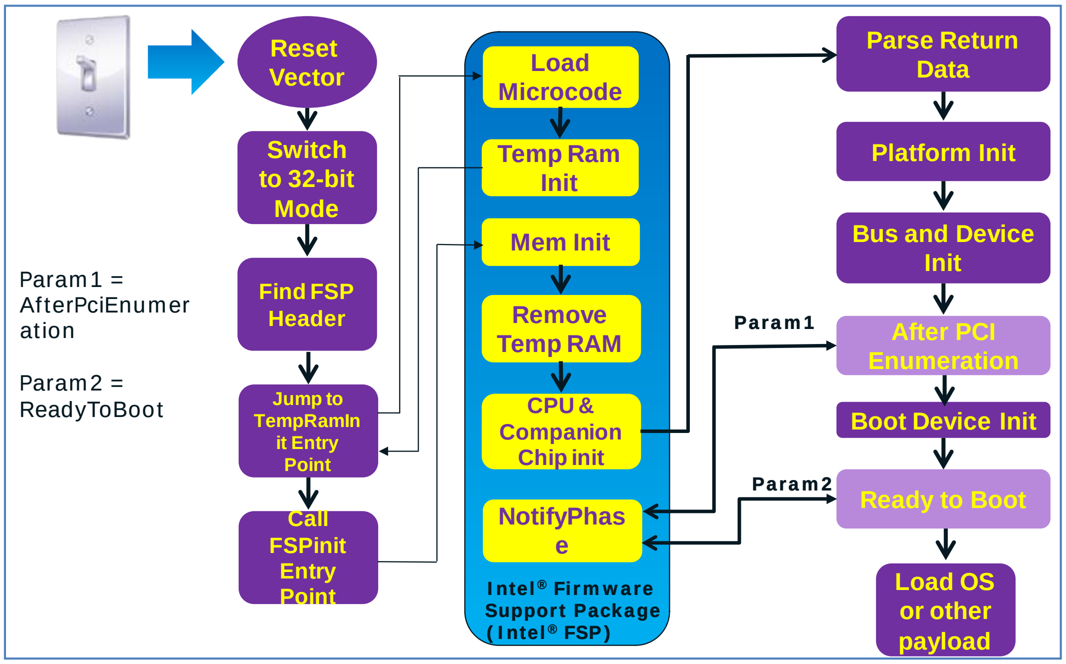
3.1 Assumptions Used in this Document

Because the Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure 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 strings that will help match the FSP Binary to a supported hardware configuration. For the Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure FSP, the Image Id will be "AVN-FSP"
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® Atom™ Processor C2000 Product Family for Communications Infrastructure FSP, the default ImageBase is 0xFFF80000.
32	4	ImageAttribute	Attributes of the FSP binary.
36	4	CfgRegionOffset	Offset of the configuration region. This offset is relative to the FSP binary base address.

Byte Offset	Size in Bytes	Field	Description
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/specs/

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.
- Refer to Appendix B for a sample code snippet that 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® Atom™ Processor C2000 Product Family for Communications Infrastructure 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

- The offset of the TempRamInitEntry can be found at **0xFFF800C4**
- The offset of the FspInitEntry can be found at **0xFFF800C8**
- 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 Data Structure Convention

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

6.3 Entry-Point Calling Convention

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

- All parameters are pushed onto the stack in 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. Refer to individual API description for any special notes and these exceptions.

6.4 Exit Convention

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

The FspInit and the FspNotify interfaces preserves all the general purpose registers except "eax". The return status is passed back through the eax register.

The FSP reserves some memory for its internal use and the memory region that is used by the FSP is passed back through a HOB. This is a generic resource HOB, but the owner field of the HOB will identify the owner as FSP. Refer to Section 7 FSP Output for more details. The boot loader is expected 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.

6.5 TempRamInitEntry

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

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

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

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

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

6.5.1 Prototype

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

6.5.2 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 tries to load all the microcodes that it finds in the region that are compatible with the silicon it is supporting. This microcode region is remembered by FSP so that it can be used to load microcode for all APs later on during the FspInit API call.

6.5.3 Related Definitions

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

6.5.3.1 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 return 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.

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

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

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

6.5.3.2 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 in the header
  add   0x1c(%ebp), %eax   # add 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
  #
/* EAX - Return value, defined in
src/mainboard/intel/mohonpeak/fsptypes.h */
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 early_init to initialize meomry and chipset.
Pass the FSP INFO
  # Header address as a paramater
  #
  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


/** C Based Basic Initialization
 *
 * Platform configuration with Temp Stack starts here.
 *
 */
void early_init (FSP_INFO_HEADER *fsp_info)
{
  .
  .
  .
  //
  // Call FspInit API
  //
  .
  .
  .
}
```

6.5.4 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 ECX and EDX.

6.6 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 does not return to the boot loader from where it was called, but instead returns control to the boot loader by calling the continuation function.

6.6.1 Prototype

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

6.6.2 Parameters

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

6.6.3 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 structure, including dependent structures, for the Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure FSP is defined below.

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

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

typedef struct {
    CONST MEM_DOWN_DIMM_CONFIG *MemDownDimmConfig[2][2];
} FSP_INIT_RT_PLATFORM_BUFFER;
```

See Appendix C for further details on the MEM_DOWN_DIMM_CONFIG structure.

6.6.4 Return Values

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

6.6.5 Sample Code

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

typedef struct {
  void                *NvsBufferPtr;
  void                *RtBufferPtr;
  CONTINUATION_PROC   ContinuationFunc;
} FSP_INIT_PARAMS;

typedef struct {
  UINT32            *StackTop;
  UINT32             BootMode;
} FSP_INIT_RT_COMMON_BUFFER;

typedef struct {
  FSP_INIT_RT_COMMON_BUFFER  Common;
} FSP_INIT_RT_BUFFER;

#define FSPAPI __attribute__((cdecl))
typedef FSP_STATUS (FSPAPI *FSP_FSP_INIT)    (FSP_INIT_PARAMS
*FspInitParamPtr);

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;

  memset((void*)&FspRtBuffer, 0, sizeof(FSP_INIT_RT_BUFFER));
  FspRtBuffer.Common.StackTop = &_stack_top;
  FspInitParams.NvsBufferPtr = get_NVRAM_ptr();
  FspRtBuffer.Common.BootMode = get_BootMode();
  FspInitParams.RtBufferPtr  = (FSP_INIT_RT_BUFFER
*)&FspRtBuffer;
  FspInitParams.ContinuationFunc =
(CONTINUATION_PROC)ContinuationFunc;
  FspInitApi = (FSP_FSP_INIT)(fsp_info->ImageBase + fsp_info-
>FspInitEntry);
  FspInitApi(&FspInitParams);

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

void ContinuationFunc (EFI_STATUS Status, VOID *HobListPtr)
{

    /* Update global variables */
    FspHobListPtr = HobListPtr;

    __asm__ __volatile__ (
        "movl  %%ebx, %0\n\t" /* The FSP_INFO_HEADER is saved in
EBX after TempRamInit API call*/
   : "=r"(fsp_info_header));

    /* Continue the boot */
    advancedInit ();

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

6.6.6 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 stack top where the boot loader wants to establish the stack after memory is initialized and available for use.

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

This FspInit API initializes the permanent memory and switches the stack from the temporary memory to the permanent memory as specified by StackTop. Sometimes switching stack in a function can cause some unexpected execution results because the compiler is not aware of the stack change during runtime and the precompiled code may still refer to the old stack for data and pointers. A stack switch, therefore, requires assembly code to patch the data for the new stack location, which may lead to compatibility issues. To avoid such possible compatibility issues introduced by different compilers and to ease the integration of FSP with a boot loader, the API uses the "ContinuationFunction" parameter to continue the boot loader execution flow rather than returning as a normal C function. Although this API is called as a normal C function, it never returns.

The FSP needs to get some parameters from the boot loader when it initializes the silicon. These parameters are passed from the boot loader to the FSP through the RtBuffer structure pointer.

The FSP returns a data structure which must be saved in a non-volatile memory such as SPI flash and the boot loader must pass the pointer to this structure (through NvsBufferPtr) back to the FspInit API upon every initialization.

Note: S3 is not supported on Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure. Therefore, the BootMode parameter for Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure FSP currently supports BOOT_WITH_FULL_CONFIGURATION only.

The BootMode parameter is used by the boot loader to let FSP know what boot path the platform is taking. It is used in conjunction with the data passed in via NvsBufferPtr, to cater for modes such as S3 resume.

During execution the FSP builds 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.

6.7 NotifyPhaseEntry

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

The FSP locks 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.

6.7.1 Prototype

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

6.7.2 Parameters

NotifyPhaseParamPtr	Address pointer to the **NOTIFY_PHASE_PRAMS**

6.7.2.1 Related Definitions

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

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

`EnumInitPhaseAfterPciEnumeration`

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

`EnumInitPhaseReadyToBoot`

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

6.7.3 Return Values

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

6.7.4 Sample Code

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

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

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

  typedef struct {
    FSP_INIT_PHASE    Phase;
  } NOTIFY_PHASE_PARAMS;

  void FspNotifyPhase (uint32_t Phase)
{
    FSP_NOTFY_PHASE        NotifyPhaseProc;
    NOTIFY_PHASE_PARAMS    NotifyPhaseParams;
    FSP_STATUS             Status;

    /* call FSP PEI to Notify PostPciEnumeration */
    NotifyPhaseProc    = (FSP_NOTFY_PHASE)(fsp_info_header-
>ImageBase + fsp_info_header->NotifyPhaseEntry);
    NotifyPhaseParams.Phase = Phase;
    Status = NotifyPhaseProc (&NotifyPhaseParams);
    if (Status != 0) {
        printf("FSP API NotifyPhase failed for phase %d!\n",
Phase);
    }
}
```

§

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/specs/

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

The section below describes the GUID HOBs that are produced by the FSP. GUID HOB structures are non-architectural in the sense that the structure 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.

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 that might be useful for the boot loader even after the FspInit call. So before destroying the temporary memory, all contents in this sub region are migrated to the permanent memory. FSP builds a boot loader temporary memory data HOB, which it can use 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 FSP Reserved Memory Resource Descriptor HOB

The FSP reserves some memory for its internal use and a descriptor for this memory region used by the FSP is passed back though a HOB. This is a generic resource HOB, but the owner field of the HOB identifies the owner as FSP.

```
#define FSP_HOB_RESOURCE_OWNER_FSP GUID \
{ 0x69a79759, 0x1373, 0x4367, { 0xa6, 0xc4, 0xc7, 0xf5, 0x9e,
0xfd, 0x98, 0x6e } }
```

7.3 Non-Volatile Storage HOB

```
#define FSP_NON_VOLATILE_STORAGE_HOB_GUID \

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

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

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

7.4 HOB Sample Code

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

7.4.1 Hob Infrastructure Sample Code

Refer to Appendix A for sample code.

7.4.2 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
GetFspReservedMemoryFromGuid (
  UINT32          *FspMemoryBase,
  UINT32            *FspMemoryLength,
  EFI_GUID          FspReservedMemoryGuid
  )
{
  EFI_PEI_HOB_POINTERS Hob;
  //
  // Get the HOB list for processing
  //
  Hob.Raw = GetHobList();
  *FspMemoryBase = 0;
  *FspMemoryLength = 0;
  //
  // Collect memory ranges
  //
  while (!END_OF_HOB_LIST (Hob)) {
    if (Hob.Header->HobType == EFI_HOB_TYPE_RESOURCE_DESCRIPTOR)
{
       if (Hob.ResourceDescriptor->ResourceType ==
EFI_RESOURCE_MEMORY_RESERVED) {
        if (CompareGuid(&Hob.ResourceDescriptor->Owner,
&FspReservedMemoryGuid)) {
          *FspMemoryBase = (UINT32) (Hob.ResourceDescriptor-
>PhysicalStart);
          *FspMemoryLength = (UINT32) (Hob.ResourceDescriptor-
>ResourceLength);
          break;
        }
      }
    }
    Hob.Raw = GET_NEXT_HOB (Hob);
  }
  return;
}
```

7.4.3 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 that is used by the FSP during the initialization. The configurable data region has two sets of data:

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

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

In addition to static configuration, the UPD data can be overridden by the boot loader during runtime. The UPD data is organized as a structure. The FspInit API parameter includes an UpdDataRgnPtr pointer, which can be initialized to point to the UPD data structure. If this pointer is initialized to NULL when calling the FspInit API, the FSP uses the default built-in UPD configuration data in the FSP binary. However, if the boot loader wants to override any of the UPD parameters, it has to copy the whole UPD structure from flash to memory, override the parameters, initialize the UpdDataRgnPtr pointer to the address of the UPD structure with updated data in memory, and call FspInit API. The FSP uses this data structure instead of the default configuration region data for platform initialization. The UPD data structure pointed by pointer UpdDataRgnPtr is a project-specific structure; refer to Section 8.1 for the details of this structure.

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 sets up a new stack in the permanent memory. However, the FSP saves the stack that was in the Temporary Memory in a HOB. If the boot loader wants 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 definitions are provided in the file fspvpd.h, which comes in the FSP release package. To update these configuration options statically using the BCT, a BSF file is required. This file contains the detailed information on all configurable options, including description, help information, valid value range, and the default value.

Refer to the RangeleyFsp.bsf file in the release package for more information.

8.1 VPD/UPD Data Structure

As stated above, the VPD/UPD data structure and related structure definitions are provided in the file fspvpd.h. The basic information for each option is provided in the BCT configuration file RangeleyFsp.bsf. The user can use the BCT tool to load this BSF file to get the detailed configuration option information.

8.1.1 VPD Data Region

This VPD_DATA_REGION region can only be configured statically by the BCT tool. Only very limited options in this region can be configured. Most of the configurable options are provided in the UPD data region.

Below is some additional information for some of the fields in VPD_DATA_REGION.

`PcdVpdRegionSign`

This field is not an option. It is a signature for the VPD data region. It can be used by the boot loader to validate the VPD region. This field will not change across different FSP releases for the same silicon.

`PcdImageRevision`

This field is not an option. It is a revision ID for the FSP release. It can be used by the boot loader to validate the VPD/UPD region. If the value in this field is changed for a FSP release, the boot loader should not assume the same layout for UPD_DATA_REGION/VPD_DATA_REGION data structure. Instead it should use the new fspvpd.h coming with the FSP release package.

`PcdUpdRegionOffset`

This field is not an option. It contains the offset of the UPD data region within the FSP release image. The boot loader can use it to find the location of UPD_DATA_REGION. Refer to Appendix D – Sample Code to Find FSP UPD_DATA_REGION.

`PcdFspReservedMemoryLength`

This option is used to specify the reserved memory size for the FSP usage. FSP will consume certain memory resource during the initialization and this memory range must be reserved. This range will be reported through the GUIDed HOB mentioned in Section 7.2. In most of the cases it does not need change.

`PcdSpdWriteProtect`

Used to enable/disable SPD Write Protect. Disabling this protection allows writes to slaves 0xA0 – 0xAE. After completing SPD flow, it is highly recommended that the Boot Loader set SPD_WD. This will prevent any future Writes (accidental or malicious) to the DIMM EEPROMs: a malicious Write could corrupt an EEPROM and cause a Permanent Denial of Service (platform won't boot) until the DIMM is replaced. A platform may be designed such that SMBus0 segment includes EEPROMs, other than DIMM EEPROM. (Not Intel recommended.) In this case, if the ODM expects Write access by a regular SW driver or by BIOS/SMM at a later time, it is possible to leave SPD_WD cleared. However, the ODM should know that this exposes the DIMM EEPROMs to the security vulnerability noted above.

8.1.2 UPD Data Region

This UPD_DATA_REGION region can be configured statically by the BCT tool in the same way as VPD data region. However, this region can also be overridden by the boot loader at runtime. It provides more flexibility for the boot loader to change the options dynamically basing on certain conditions.

The fields in UPD_DATA_REGION are described below.

`Signature`

This field is not an option. It is a signature for the UPD data region. It can be used by boot loader to validate the UPD region. The boot loader should never override this field.

`PcdPrintDebugMessages`

Used to enable/disable printing of some debug messages during FSP execution.

`PcdEnableLan`

Used to enable/disable the LAN controller on the SOC.

`PcdMrcRmtSupport`

Used to enable/disable MRC Rank Margin Tool. If enabled, the MRC prints out the rank margining information so that it can be used as the input for the Rank Margin Tool (RMT) to analyze the platform memory sub-system margining. To enable it PcdFastBoot option should be disabled because MRC fast boot path skips normal memory training steps. To get the RMT log, you must also enable Serial Debug Messages.

`PcdMrcRmtCpgcExpLoopCntValue`

Used to configure Rank Margin Tool (RMT). For setting the CPGC exp_loop_cnt field for RMT execution refer to document #535399 *RMT User Guide*.

`PcdMrcRmtCpgcNumBursts`

Used to configure Rank Margin Tool (RMT). For setting the CPGC num_bursts field for RMT execution refer to document #535399 *RMT User Guide*.

`PcdExtendedTemperatureEnable`

If DIMM supports Extended Temperature Range, then enabling this PCD will automatically configure the refresh rate to be double refresh rate.

If DIMM does not support Extended Temperature Range, then enabling this PCD will have no effect.

`PcdFastboot`

Used to enable/disable fast boot path in MRC. Once enabled, all following boots will use the presaved MRC data to improve the boot performance.

`PcdSerialPortBaudRate`

Configure the serial port baud rate for the FSP binary to one of eight predefined standard settings. The default baud rate is 115200.

`PcdCustomerRevision`

Customer can add a label/version ID, max. 32 characters. This label can be retrieved by the boot loader, see Appendix D.

`PcdMemoryDown`

For the design with memory down implemented on the board enable this PCD. If it is disabled, it indicates the board uses the normal DIMMs and MRC will use the standard mechanism to read the SPD data from the DIMMs. Otherwise, it indicates the board uses hardcoded SPD data. Refer to Appendix C– Memory Down Configuration.

`PcdRegionTerminator`

This field is not an option. It is a terminator to mark the end of the data structure. The boot loader should never override this field.

`Reserved/Unused`

UPD_DATA_REGION may contain some reserved or unused fields in the data structure. For these fields it is required to keep the original default values provided in the FSP binary. Intel always recommends copying the whole UPD_DATA_REGION from the flash to local structure in stack before overriding any field.

§

9 Tools

A Binary Configuration Tool (BCT) is provided with the FSP binary that can be used on the FSP binary to allow a user to modify certain well defined configuration values in the FSP binary. The BCT typically provides a graphical user interface (GUI). The Binary Configuration Tool (BCT) is 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 is not included in Intel® FSP.

10.2 Bus Enumeration

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

10.3 Security

Intel® FSP does not provide security features besides making them 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, contact the local 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 2.0 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
  /// 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 HO
  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 wil
  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 Nam
  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 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");

#ifdef __PRE_RAM__
  __asm__ __volatile__ (
      ".global find_fsp_entry\n\t"
      "find_fsp_entry:\n\t"
    );
#endif

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

  //
  // Validate FV signature _FVH
  //
  if (((EFI_FIRMWARE_VOLUME_HEADER *)ptr)-> Signature !=
EFI_FVH_SIGNATURE) {
    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 end of FV header address to 8 bytes
  //
  ptr  = (UINT8 *)(((UINT32)ptr + 7) & 0xFFFFFFF8);

  //
  // Now ptr is pointing to thr FFS Header. Verify if the GUID
  // matches the FSP_INFO_HEADER GUID
  //
  if (guidcompare(((EFI_GUID) (((EFI_FFS_FILE_HEADER *)ptr)-
>Name)), ((EFI_GUID) gFspInfoHeaderGuid))) {
      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");
}
```

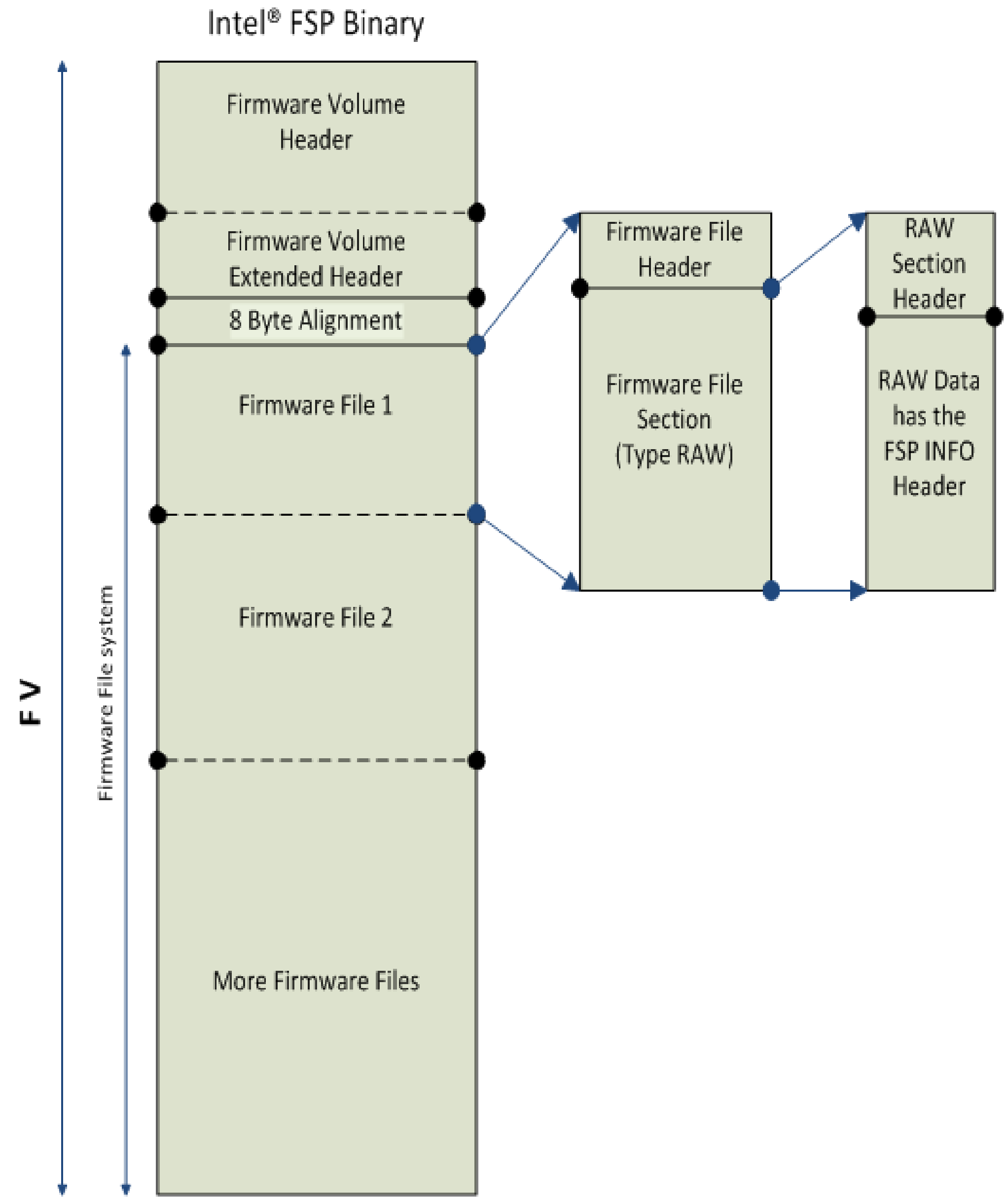
Now, call this function using a temporary ROM stack containing the return address and bypass the prolog or epilog code of the C function like below.

```
  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 Memory Down Configuration

For memory down configurations, ensure that the MemoryDown PCD is enabled via the BCT tool.

Then the following input structure should be filled in for each memory down DIMM prior to calling the FspInit API.

```
typedef struct {
     UINT8   DRAMDeviceType;                 // 2   DRAM Device Type
     UINT8   ModuleType;                     // 3   Module Type
     UINT8   SDRAMDensityAndBanks;           // 4   SDRAM Density
                                             //     and Banks
     UINT8   SDRAMAddressing;                // 5   SDRAM Addressing
     UINT8   VDD;                            // 6   Module Nominal
                                             //     Voltage
     UINT8   ModuleOrganization;             // 7   Module
                                             //     Organization
     UINT8   ModuleMemoryBusWidth;           // 8   Module Memory
                                             //     Bus Width
     UINT8   TimebaseDividend;               // 10  Medium Timebase
                                             //     (MTB) Dividend
     UINT8   TimebaseDivisor;                // 11  Medium Timebase
                                             //     (MTB) Divisor
     UINT8   SDRAMMinimumCycleTime;          // 12  SDRAM Minimum
                                             //     Cycle Time
                                             //     (tCKmin)
     UINT8   CASLatenciesLSB;                // 14  CAS Latencies
                                             //     Supported, Least
                                             //     Significant Byte
     UINT8   CASLatenciesMSB;                // 15  CAS Latencies
                                             //     Supported, Most
                                             //     Significant Byte
     UINT8   MinimumCASLatencyTime;          // 16  Minimum CAS
                                             //     Latency Time
                                             //     (tAAmin)
     UINT8   MinimumWriteRecoveryTime;       // 17  Minimum Write
                                             //     Recovery Time
                                             //     (tWRmin)
     UINT8   MinimumRASToCASDelayTime;       // 18  Minimum RAS# to
                                             //     CAS# Delay Time
                                             //     (tRCDmin)
```

```
  UINT8  MinimumRowToRowDelayTime;      // 19  Minimum Row
                                        //     Active to Row
                                        //     Active Delay
                                        //     Time (tRRDmin)
 UINT8  MinimumRowPrechargeDelayTime;   // 20  Minimum Row
                                        //     Precharge Delay
                                        //     Time (tRPmin)
  UINT8  UpperNibblesFortRASAndtRC;     // 21  Upper Nibbles
                                        //     for tRAS and tRC
  UINT8  tRASmin;                       // 22  Minimum Active
                                        //     to Precharge
                                        //     Delay Time
                                        //     (tRASmin), Least
                                        //     Significant Byte
  UINT8  tRCmin;                        // 23  Minimum Active
                                        //     to Active/
                                        //     Refresh Delay
                                        //     Time (tRCmin),
                                        //     Least
                                        //     Significant Byte
  UINT8  tRFCminLeastSignificantByte;   // 24  Minimum Refresh
                                        //     Recovery Delay
                                        //     Time (tRFCmin),
                                        //     Least
                                        //     Significant Byte
  UINT8  tRFCminMostSignificantByte;    // 25  Minimum Refresh
                                        //     Recovery Delay
                                        //     Time (tRFCmin),
                                        //     Most Significant
                                        //     Byte
  UINT8  tWTRmin;                       // 26  Minimum Internal
                                        //     Write to Read
                                        //     Command Delay
                                        //     Time (tWTRmin)
  UINT8  tRTPmin;                       // 27  Minimum Internal
                                        //     Read to
                                        //     Precharge
                                        //     Command Delay
                                        //     Time (tRTPmin)
  UINT8  UpperNibbleFortFAW;            // 28  Upper Nibble for
                                        //     tFAW
  UINT8  tFAWmin;                       // 29  Minimum Four
                                        //     Activate Window
                                        //     Delay Time
                                        //     (tFAWmin)
  UINT8  SdramThermalRefreshOption;     // 31
                                      SdramThermalRefreshOption
  UINT8  ModuleThermalSensor;           // 32
                                        //  ModuleThermalSensor
  UINT8  SDRAMDeviceType;               // 33  SDRAM Device
                                        //     Type
  UINT8  tCKminFine;                     // 34  Fine Offset for
                                         //     SDRAM Minimum
                                         //     Cycle Time
                                         //     (tCKmin)
  UINT8  tAAminFine;                   // 35  Fine Offset for
                                       //     Minimum CAS
                                       //     Latency Time
                                       //     (tAAmin)
  UINT8  MACCount;                     // 41  Maximum Activate
                                       //     Count
  UINT8  ReferenceRawCardUsed;         // 62  Reference Raw Card
                                       //     Used
  UINT8  AddressMappingEdgeConnector; // 63  Address Mapping
                                       //     from Edge
                                       //     Connector to DRAM
  UINT8  ModuleManufacturerIdCodeLsb;  // 117 Module
                                       //     Manufacturer ID
                                       //     Code, Least
                                       //     Significant Byte
  UINT8  ModuleManufacturerIdCodeMsb;  // 118 Module
                                       //     Manufacturer ID
                                       //     Code, Most
                                       //     Significant Byte
  UINT8  ModuleManufacturingLocation;  // 119 Module
                                       //     Manufacturing
                                       //     Location
  UINT8  ModuleManufacturingDateYear;  // 120 Module
                                       //     Manufacturing
                                       //     Date Year
  UINT8  ModuleManufacturingDateWW;    // 121 Module
                                       //     Manufacturing
                                       //     Date creation
                                       //     work week
  UINT8  ModuleSerialNumberA;          // 122 Module Serial
                                       //     Number A
  UINT8  ModuleSerialNumberB;         // 123 Module Serial
                                       //     Number B
 UINT8  ModuleSerialNumberC;          // 124 Module Serial
                                       //     Number C
  UINT8  ModuleSerialNumberD;          // 125 Module Serial
                                       //     Number D
  UINT8  DramManufacturerIdLsb;        // 148 DRAM
                                       //     Manufacturer ID
                                       //     Code, LSB
  UINT8  DramManufacturerIdMsb;        // 149 DRAM
                                       //     Manufacturer ID
                                       //     Code, MSB
} MEM_DOWN_DIMM_SPD_DATA;
```

```
typedef struct {
  UINT32   MemoryDownDimmPopulation;  // 0 - Empty, 1 – DIMM
                                      //     populated
  MEM_DOWN_DIMM_SPD_DATA   MemoryDownDimmSpdData;
} MEM_DOWN_DIMM_CONFIG;
```

§

Appendix D Sample Code to Find FSP UPD_DATA_REGION

The sample code below will locate the UPD_DATA_REGION in the FSP binary and copy the default data to a structure.

```
void early_init (FSP_INFO_HEADER *fsp_info)
{
  FSP_FSP_INIT            FspInitApi;
  FSP_INIT_PARAMS         FspInitParams;
  FSP_INIT_RT_BUFFER      FspRtBuffer;
  VPD_DATA_REGION         *FspVpdRgn;
  UPD_DATA_REGION         FspUpdRgn;
  …
  …
  memset((void*)&FspRtBuffer, 0, sizeof(FSP_INIT_RT_BUFFER));
  FspRtBuffer.Common.StackTop = (uint32_t *) ROMSTAGE_STACK;
  FspRtBuffer.Common.UpdDataRgnPtr = (UPD_DATA_REGION
*)&FspUpdRgn;
  …
  …
  …

  /* Get VPD region start */

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


  /* Verify the VPD data region is valid */
  ASSERT((FspVpdRgn->PcdImageRevision == VPD_IMAGE_REV) &&
             (FspVpdRgn->PcdVpdRegionSign == VPD_IMAGE_ID));

  /* Copy default data from Flash */
  memcpy ((void*)&FspUpdRgn, (void *)(fsp_info->ImageBase +
               FspVpdRgn->PcdUpdRegionOffset),
sizeof(UPD_DATA_REGION));

  /* Verify the UPD data region is valid */
  ASSERT(FspUpdRgn.PcdRegionTerminator == 0x55AA);


  /* Override any UPD setting if required */
  //
  // Uncomment the line below to disable LAN device
 //
  //FspUpdRgn.PcdEnableLan = 0;
```

```
    …
    …
    …
}
```

§

Appendix E Port80 POST Codes

This appendix lists the port80 post codes that may be output during calls to FSP APIs.

E.1 TempRamInit

Post Code	Description
0x03	Microcode Loaded
0x04	Basic Init Complete

E.2 FspInit

Post Code	Description
0x12	Enable Clock Gating
0x13	Clear Self Refresh
0x14	Oem Track Init Complete
0x42	Program DDR Timing and Control Registers
0x44	Program Burst Length Mode Registers
0x45	Enable DDR Low Voltage
0x50	Handle DDRIO Phy Initialization
0xA0	MMRC SFR Vol Sel
0xA1	MMRC PLL Init
0xA2	MMRC DDR Static Init 2
0xA5	MMRC DDR Static Init Perf
0xA6	MMRC DDR Static Pwr Clk Gating
0xA7	MMRC DLL Init
0xA8	MMRC Comp Init 1
0xAA	MMRC Comp Init 2

Post Code	Description
0xAE	MMRC HMC Init
0xAF	MMRC Wr Pointer Init
0xB0	MMRC IO BUF ACT Init
0xB1	MMRC Pre Jedec Init
0xB2	MMRC DDR3 Reset
0x51	Program DDRIO CKE for each rank
0x61	Program Timing and Control registers for Memory Controller. DRP, DMAP...etc
0x62	Program Memory Mapping Registers
0x71	Perform D-unit Wake
0x81	Set Configuration for Pre-Jedec Init
0x82	Program the Jedec init for a row of memory
0x83	Program misc registers to be set after Mem Init
0x90	Program the PMI to be released by Bunit
0x91	Disable Pmi
0x92	Disable B-unit Cache
0x92	Set DDR Initialization Complete
0xB0	Handle Rank Overrides... Rank to rank switching enabled
0xB1	Enable Diffamp And Odt Overrides
0xB2	Early Set Write Vref... Vref Set: 0x40
0xB3	Program the PMI to be released by Bunit
0xB4	Enable Pmi
0xB5	Handle Memory Training
0xB0	Receive Enable.
0xD0	Rank To Rank Sequence
0xB2	Early Mpr Read.
0xB3	Fine Write Leveling
0xB4	Coarse Write Leveling

Post Code	Description
0xB6	Read Vref
0xB7	Read Training
0xB9	Write Vref. Max Vref Center.
0xBA	Set Common Vref
0xBB	Write Training
0xBC	Command Clock Training
0xBD	Command Clock Restore
0xBE	Performance Setting
0xBF	Phy View Table
0xC0	Rank Margin Tool
0xC1	Rank To Rank Pointer Offset
0xB6	Vref Override
0xBE	Enable B-unit Cache
0xC0	Reconfigure DRP
0xD1	Program DDR Timing control registers
0xE1	Configure Scrambler
0xE2	Set Periodic Resistive Compensation
0xE3	Set DDR Initialization Complete
0xE4	PerformWake
0xE5	Change Refresh Period
0xE6	Program the PMI to be released by Bunit
0xE8	Enable Pmi
0xE7	Modify ECC Bytelane
0xE9	HandlePostTraining
0xEA	PrintDunitTable
0xEB	Print MRS Table
0xEC	Pass Gate Config

Post Code	Description
0xED	Pass Gate Test
0xF0	MemoryTest Setup
0xF1	Memory Testing
0xF2	Scrub Memory
0xF3	Program PMI to be owned by Bunit
0xF4	Set Init Done
0xF5	Clear DDR3 Reset
0xF6	Program Bunit Performance Settings
0xF7	Disable HPET

§