Intel® Firmware Support Package for Intel® Atom™ Processor E3800 Product Family

Integration Guide

May 2015

Contents

Figures

Tables

Revision History

Date	Revision	Description
May 2015	1.8	Updated for Gold 4 release.
July 2014	1.7	Updated for Gold 3 release.
March 2014	1.6	First public release.
November 2013	1.5	Updated to add configuration options to Section 8.
October 2013	1.4	Updated for Gold release.
September 2013	1.3	Updated for Beta 2 release.
August 2013	1.2	Updated for Beta release.
March 2013	1.1	Updated Related Documents section.
March 2013	1.0	Initial release.

§

1 Introduction

1.1 Purpose

The purpose of this document is to describe the steps required to integrate the Intel® FSP for the Intel® Atom™ Processor E3800 Product Family (formerly Bay Trail SoC) 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 http://www.intel.com/fsp
- Binary Configuration Tool for Intel® Firmware Support Package – available at http://www.intel.com/fsp

1.5 Conventions

To illustrate some of the points better, the document will use code snippets. The code snippets follow **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
TXE	Trusted Execution Engine/Environment

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. When secure boot is enabled, the FSP will be required to be placed at a fixed address in the memory space. Refer to the Secure Boot Readme document for details.

- **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

The Intel® FSP for Intel® Atom™ Processor E3800 Product Family is built with a preferred base address of 0xFFFC0000 and so the reference code provided in the document assumes that the FSP is placed at 0xFFFC0000 after the final boot loader build.

3.2 FSP Image Revision

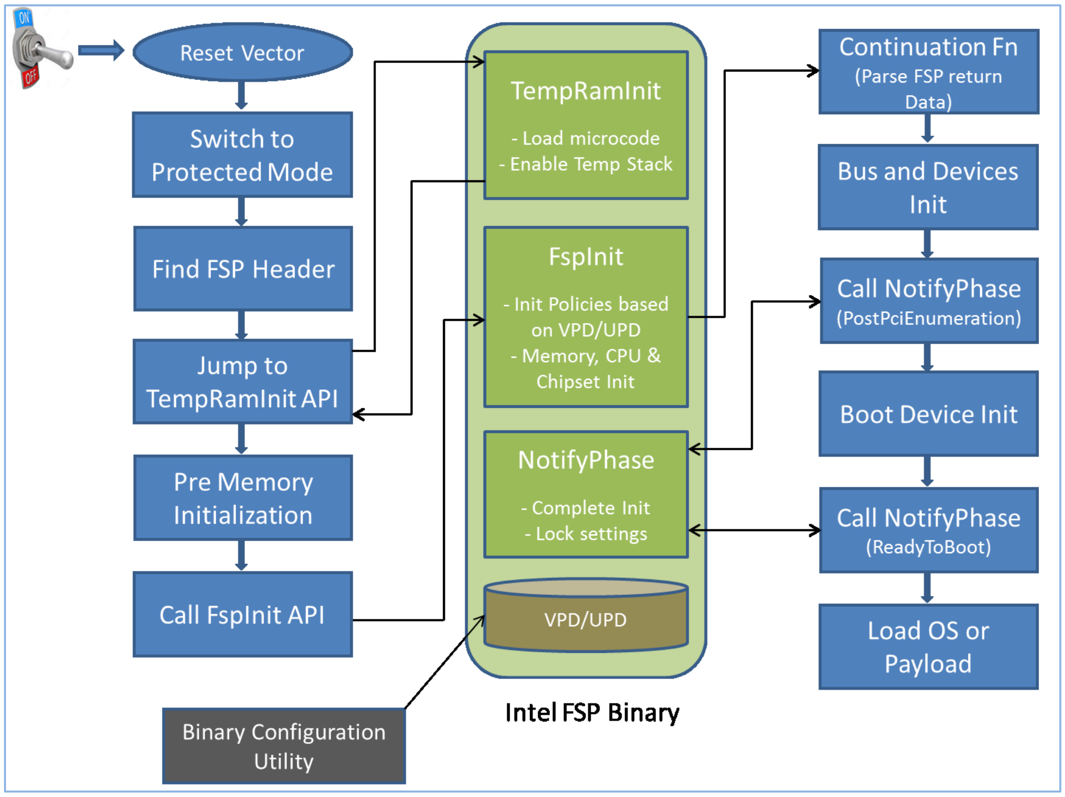
To FSP information header contains an Image Revision field that provides the revision of the FSP binary. This is an important field to consider while integrating the FSP as API parameters could change over FSP revisions. The FSP API parameters documented in this integration guide are applicable for the Image Revisions specified in this document.

The current version of the Bay Trail FSP is Gold 04 and the ImageRevision in the FSP Header is 00000304.

4 Boot Flow

Figure 1 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.

Figure 1. Boot Flow



5 FSP Binary Format

The FSP is distributed in binary format. The FSP binary contains an FSP specific **FSP_INFO_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.

Table 1. FSP Header

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. For Intel® FSP for Intel® Atom™ Processor E3800 Product Family, the ImageId is "VLYVIEW0"
24	4	ImageSize	Size of the entire FSP Binary.
28	4	ImageBase	FSP binary preferred base address. If the FSP binary is located at an address different from the preferred address, the rebasing tool is required to relocate the base before the FSP binary integration.
32	4	ImageAttribute	Attributes of the FSP binary. This field is not currently used.
36	4	CfgRegionOffset	Offset of the configuration region. This offset is relative to the FSP binary base address.

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/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_INFO_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_INFO_HEADER** structure.
- Please refer 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® FSP for Intel® Atom™ Processor E3800 Product Family, the FSP header is placed at an offset of **0x94.** The FSP lies outside the IBB area. So, when secure boot is enabled, there will be an additional step to verify the header itself by comparing it with a copy of the header inside the IBB. Refer to the Secure Boot Readme file for details.

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 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.

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

The FSP reserves some memory for its internal use and the memory region that is used by the FSP is passed back though a HOB. This is a Generic Resource HOB, but the owner field of the HOB will identify the owner as FSP. Please refer to the FSP Output section 7 for more details. The boot loader is expected to not 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 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 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 will try to load all the microcodes that it finds in the region that is compatible with the silicon it is supporting. This microcode region is remembered by FSP so that it can be used to load microcode for all APs later on during the FspInit API call.

6.5.3 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 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.

Table 2. Return Values

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

6.5.3.2 Sample Code

```
.global basic_init
basic_init:
  .
  .
  .

  #
  # Parse the FV to find the FSP INFO Header
  #
  # If secure boot is enabled, verify if the header matches the
  # signed header in IBB.
  # Please refer the secure boot readme for details.
  #
  lea   findFspHeaderStack, %esp  # _fsp_rom_start = 0xfffc0000
  lea   _fsp_rom_start, %eax
  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
  #
  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   0xfff00000      # Code Region Base
  .long   0x00100000      # 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 information header and then from the header finds the offset of the TempRamInit function. It then converts the offset to an absolute address by adding the base of the FSP binary and jumps to the TempRamInit function.

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

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

6.6 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 defined below has a pointer to a data structure called UPD Data. Refer to Section 8 for details.

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

typedef struct {
  FSP_INIT_RT_COMMON_BUFFER          Common;
 } FSP_INIT_RT_BUFFER;

typedef struct {
  UINT32    VendorDeviceId;
  UINT16    SubSystemId;
  UINT8     RevisionId;     /// 0xFF applies to all
                              /// steppings
  UINT8     FrontPanelSupport;
  UINT16    NumberOfRearJacks;
  UINT16    NumberOfFrontJacks;
} PCH_AZALIA_VERB_TABLE_HEADER;
typedef struct {
  PCH_AZALIA_VERB_TABLE_HEADER  VerbTableHeader;
  UINT32                        *VerbTableData;
} PCH_AZALIA_VERB_TABLE;
```

```
typedef struct {
  UINT8    Pme        : 1;  /// 0: Disable; 1: Enable
  UINT8    DS         : 1;  /// 0: Docking not supported;
                             /// 1:Docking supported
  UINT8    DA         : 1;  ///  0: Docking not attached;
                             ///  1:Docking attached
  UINT8    HdmiCodec : 1;  /// 0: Disable; 1: Enable
  UINT8    AzaliaVCi : 1;  /// 0: Disable; 1: Enable
  UINT8    Rsvdbits  : 3;
  UINT8    AzaliaVerbTableNum;    /// # of verb tables
  PCH_AZALIA_VERB_TABLE  *AzaliaVerbTable; /// Ptr to
                                            /// the
                                            /// actual
                                            /// verb
                                            /// table arrays
  UINT16   ResetWaitTimer;    /// The delay in uS after
                                    /// Azalia reset
} PCH_AZALIA_CONFIG;
```

6.6.4 Return Values

Table 3. 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

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

const UINT32 mAzaliaVerbTableData13[] = {
    //
    //ALC262 Verb Table - 10EC0262
    //
    //Pin Complex (NID 0x11 )
    0x01171CF0,
    0x01171D11,
    0x01171E11,
    0x01171F41,
    //Pin Complex (NID 0x12 )
    0x01271CF0,
    0x01271D11,
    0x01271E11,
    0x01271F41,
    //Pin Complex (NID 0x14 )
    0x01471C10,
    0x01471D40,
    0x01471E01,
    0x01471F01,
    //Pin Complex (NID 0x15 )
    0x01571CF0,
    0x01571D11,
    0x01571E11,
    0x01571F41,
    //Pin Complex (NID 0x16 )
    0x01671CF0,
    0x01671D11,
    0x01671E11,
    0x01671F41,
    //Pin Complex (NID 0x18 )
    0x01871C20,
    0x01871D98,
    0x01871EA1,
    0x01871F01,
    //Pin Complex (NID 0x19 )
    0x01971C21,
    0x01971D98,
    0x01971EA1,
    0x01971F02,
    //Pin Complex (NID 0x1A )
    0x01A71C2F,
    0x01A71D30,
    0x01A71E81,
    0x01A71F01,
    //Pin Complex (NID 0x1B )
    0x01B71C1F,
    0x01B71D40,
    0x01B71E21,
    0x01B71F02,
    //Pin Complex (NID 0x1C )
    0x01C71CF0,
    0x01C71D11,
    0x01C71E11,
    0x01C71F41,
    //Pin Complex (NID 0x1D )
    0x01D71C01,
    0x01D71DC6,
    0x01D71E14,
    0x01D71F40,
    //Pin Complex (NID 0x1E )
    0x01E71CF0,
    0x01E71D11,
    0x01E71E11,
    0x01E71F41,
    //Pin Complex (NID 0x1F )
    0x01F71CF0,
    0x01F71D11,
    0x01F71E11,
    0x01F71F41
};

const PCH_AZALIA_VERB_TABLE mAzaliaVerbTable[] = {
    {
        //
        // VerbTable: (RealTek ALC262)
        //  Revision ID = 0xFF, support all steps
        //  Codec Verb Table For AZALIA
        //  Codec Address: CAd value (0/1/2)
        //  Codec Vendor:  0x10EC0262
        //
        {
            0x10EC0262,     // Vendor ID/Device ID
            0x0000,         // SubSystem ID
            0xFF,           // Revision ID
            0x01,           // Front panel support (1=yes, 2=no)
            0x000B,         // Number of Rear Jacks = 11
            0x0002          // Number of Front Jacks = 2
        },
        (UINT32 *)mAzaliaVerbTableData13
    }
};
```

```
const PCH_AZALIA_CONFIG mAzaliaConfig = {
    .Pme = 1,
    .DS = 1,
    .DA = 0,
    .HdmiCodec = 1,
    .AzaliaVCi = 1,
    .Rsvdbits = 0,
    .AzaliaVerbTableNum = 1,
    .AzaliaVerbTable = (PCH_AZALIA_VERB_TABLE *)mAzaliaVerbTable,
    .ResetWaitTimer = 300
};

typedef struct {
  UPD_DATA_REGION    fsp_upd_data;
} STAGE_SHARED_DATA;

void GetUpdDefaultFromFsp (FSP_INFO_HEADER *FspInfo,
UPD_DATA_REGION   *UpdData)
{
  VPD_DATA_REGION *VpdDataRgnPtr;
  UPD_DATA_REGION *UpdDataRgnPtr;
  VpdDataRgnPtr = (VPD_DATA_REGION *)(UINT32)(FspInfo-
  >CfgRegionOffset  + FspInfo->ImageBase);
  UpdDataRgnPtr = (UPD_DATA_REGION *)(UINT32)(VpdDataRgnPtr-
  >PcdUpdRegionOffset + FspInfo->ImageBase);
  memcpy_s((void*)UpdData, sizeof(UPD_DATA_REGION),
(void*)UpdDataRgnPtr, sizeof(UPD_DATA_REGION));
}

void early_init (FSP_INFO_HEADER *fsp_info, UINT32 stack_base)
{
  FSP_FSP_INIT               FspInitApi;
  FSP_INIT_PARAMS            FspInitParams;
  FSP_INIT_PARAMS           *FspInitParamsPtr;
  FSP_INIT_RT_BUFFER         FspRtBuffer;
  UINT8                      BootMode;
  STAGE_SHARED_DATA          StageSharedData;
  UINT32                     SharedDataOff;

  post_code (0x10);

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

  /* Detect Boot Mode */
  BootMode = getBootMode();

  /* Initialize the UPD Data */
  GetUpdDefaultFromFsp (fsp_info, &StageSharedData.fsp_upd_data);
  StageSharedData.fsp_upd_data.AzaliaConfigPtr = (UINT32)
&mAzaliaConfig;
```

```
  /* Initialize the FspInit Common parameters */
  FspRtBuffer.Common.BootMode = BootMode;
  FspRtBuffer.Common.UpdDataRegPtr =
&StageSharedData.fsp_upd_data;
  if (BootMode == BOOT_ON_S3_RESUME) {
    FspRtBuffer.Common.StackTop = (UINT32 *)(permanentHeapBase +
permanentHeapSize);
  } else {
    FspRtBuffer.Common.StackTop = &_stack_top;
  }

  /* Initialize the FspInit Params */
  FspInitParams.RtBufferPtr  = (FSP_INIT_RT_BUFFER
*)&FspRtBuffer;
  FspInitParams.ContinuationFunc =
(CONTINUATION_PROC)continuation_func;
  FspInitParams.NvsBufferPtr  = (void *)&_nvram_base;

  /*
   * Save fsp_info in EBX so that we can get it back in
continuation_func()
   * since FSP will preserve all registers but EAX
   */
  FspInitApi    = (FSP_FSP_INIT)(fsp_info->ImageBase + fsp_info-
>FspInitEntry);
  SharedDataOff = (UINT32)&StageSharedData - stack_base;

  FspInitParamsPtr = &FspInitParams;

  asm volatile (
        "push %0;"
        "call *%%eax;"
        :  : "m"(FspInitParamsPtr), "a"(FspInitApi),
"b"(fsp_info), "c"(SharedDataOff));

  /*
   * Should never get here. Control will continue from
continuation_func
   * This line below is to prevent the compiler from optimizing
structure intialization
   */
  FspInitApi (&FspInitParams);

  while (1);
}


void ContinuationFunc (EFI_STATUS Status, VOID *HobListPtr)
{
  /* Update global variables */
  FspHobListPtr = HobListPtr;

  /* The FSP_INFO_HEADER is saved in EBX in early_init */
  /* The Shared Data Offset is saved in ECX in early_init */
  __asm__ __volatile__ (
      "movl %%ebx, %0\n\t"
      "movl %%ecx, %1\n\t"
      : fsp_info_header, shared_data_off);

  /* Get the HOB with the Temp Stack Data*/
  GetTempRamStack(&temp_stack_hob, &hobsize);

  shared_data  = (STAGE_SHARED_DATA *)((uint32_t)temp_stack_hob +
shared_data_off);
  upd_data_ptr = &(shared_data->fsp_upd_data);

  /* Continue the boot */
  advancedInit ();

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

6.6.6 Description

One important piece of data pointed by RtBufferPtr in the FSP_INIT_PARAMS structure is the "StackTop". This passes the address of the stack top where the boot loader wants to establish the stack after memory is initialized and available for use.

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

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

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

A set of parameters that the FSP may need to initialize memory under special circumstances, such as during an S3 resume and during fast boot mode, are returned by the FSP to the boot loader during a normal boot. The boot loader is expected to store these parameters in a non-volatile memory such as 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 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 will be platform dependent and will be documented with the FSP release. Examples of boot phases include "post pci enumeration" and "ready to boot".

The FSP will lock the configuration registers to enhance security as required by the BWG when it is notified that the boot loader is ready to transfer control to the operating system.

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 completes the PCI enumeration and the resource allocation for the PCI devices is complete. FSP will use it to do some specific initialization for processor and chipset that requires PCI resource assignment.

EnumInitPhaseReadyToBoot

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

6.7.3 Return Values

Table 4. Return Values

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

6.7.4 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 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/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 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 the boot loader even after the FspInit call. So before destroying the temporary memory, all contents in this sub region will be migrated to the permanent memory, FSP builds a boot loader temporary memory data HOB and the boot loader can use it to access the data saved in the temporary memory after FspInit API if necessary. If the boot loader does not care about the previous data, 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 S3 resume.

The boot loader needs to parse the HOB list to see if such a GUID HOB exists after returning from the FspInit() API. If so, the boot loader should extract the data portion from the HOB, and then save it into a platform-specific NVS device, such as flash, EEPROM, etc. 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 Parsing Code

```
VOID
GetLowMemorySize (
  UINT32         *LowMemoryLength
  )
{
  EFI_PEI_HOB_POINTERS    Hob;

  *LowMemoryLength = 0x100000;

  //
  // Get the HOB list for processing
  //
  Hob.Raw = GetHobList();

  //
  // 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
GetHighMemorySize (
  UINT64         *HighMemoryLength
  )
{
  EFI_PEI_HOB_POINTERS    Hob;

  *HighMemoryLength = 0x0;

//
  // Get the HOB list for processing
  //
  Hob.Raw = GetHobList();

  //
  // 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 4GB to be collected here
        //
        if (Hob.ResourceDescriptor->PhysicalStart >=
(EFI_PHYSICAL_ADDRESS) 0x100000000) {
          *HighMemoryLength += (UINT64) (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;
}

VOID
GetFspNVStorageMemory (
  VOID          **FspNVStorageHob,
  UINT16      *DataSize
  )
{
  EFI_GUID                FspNVStorageHobGuid =
FSP_NON_VOLATILE_STORAGE_HOB_GUID;
  UINT8                 *GuidHob;
  EFI_HOB_GENERIC_HEADER  *GuidHobHdr;

  GuidHob = GetFirstGuidHob(&FspNVStorageHobGuid);
  if (!GuidHob) {
    *FspNVStorageHob = 0;
    *DataSize = 0;
  } else {
    *FspNVStorageHob = GET_GUID_HOB_DATA (GuidHob);
    GuidHobHdr = (EFI_HOB_GENERIC_HEADER *)GuidHob;
    *DataSize =  GET_GUID_HOB_DATA_SIZE (GuidHobHdr);
  }
}

VOID
GetTempRamStack (
  VOID         **TempRamStackPtr,
  UINT16      *DataSize
  )
{

  EFI_GUID                FspBootloaderTemporaryMemoryHobGuid =
FSP_BOOTLOADER_TEMPORARY_MEMORY_HOB_GUID;
  UINT8                 *GuidHob;
  EFI_HOB_GENERIC_HEADER  *GuidHobHdr;

GuidHob = GetFirstGuidHob(&FspBootloaderTemporaryMemoryHobGuid);
  if (!GuidHob) {
    *TempRamStackPtr = 0;
    *DataSize = 0;
  } else {
    *TempRamStackPtr = GET_GUID_HOB_DATA (GuidHob);
   GuidHobHdr = (EFI_HOB_GENERIC_HEADER *)GuidHob;
    *DataSize =  GET_GUID_HOB_DATA_SIZE (GuidHobHdr);
  }
}
```

7.5 Hob Infrastructure Sample Code

Please refer to the Appendix - A for sample code.

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.

Special attention should be paid to the AzaliaConfigPtr variable in the UPD Data structure below. This is a pointer to the PCH_AZALIA_CONFIG data structure. If the PCH_AZALIA_CONFIG data structure is in the temporary stack and AzaliaConfigPtr is pointing to an address in the temporary stack, it may be valid when FspInit API is called, but by the time FSP uses this variable, the temporary stack would have been destroyed after relocation and AzaliaConfigPtr would still be pointing to an address in the old temporary stack region and thus be pointing to an incorrect address. To avoid this problem, the PCH_AZALIA_CONFIG data structure should be declared a constant and placed uncompressed in the flash memory and AzaliaConfigPtr should be initialized to point to this address in the flash memory. This will guarantee that the AzaliaConfigPtr will still be valid when FSP uses it.

```
#pragma pack(1)
```

```
typedef struct {
  UINT8    EnableMemoryDown;
  UINT8    DRAMSpeed;       /* DRAM Speed  */
  UINT8    DRAMType;        /* DRAM Type */
  UINT8    DIMM0Enable;     /* DIMM 0 Enable */
  UINT8    DIMM1Enable;     /* DIMM 1 Enable */
  UINT8    DIMMDWidth;      /* DRAM device data width */
  UINT8    DIMMDensity;     /* DRAM device data density */
  UINT8    DIMMBusWidth;    /* DIMM Bus Width */
  UINT8    DIMMSides;       /* Ranks Per DIMM */
  UINT8    DIMMtCL;         /* tCL */
  UINT8    DIMMtRPtRCD;     /* tRP and tRCD in DRAM clk - 5:12.5ns, 6:15ns, etc. */
  UINT8    DIMMtWR;         /* tWR in DRAM clk  */
  UINT8    DIMMtWTR;        /* tWTR in DRAM clk */
  UINT8    DIMMtRRD;        /* tRRD in DRAM clk */
  UINT8    DIMMtRTP;        /* tRTP in DRAM clk */
  UINT8    DIMMtFAW;        /* tFAW in DRAM clk */
} MEMORY_DOWN_DATA;

----------

typedef struct _UPD_DATA_REGION {
  UINT64                   Signature;                 /* Offset 0x0000 */
  UINT8                    ReservedUpdSpace0[24];     /* Offset 0x0008 */
  UINT16                   PcdMrcInitTsegSize;        /* Offset 0x0020 */
  UINT16                   PcdMrcInitMmioSize;        /* Offset 0x0022 */
  UINT8                    PcdMrcInitSPDAddr1;        /* Offset 0x0024 */
  UINT8                    PcdMrcInitSPDAddr2;        /* Offset 0x0025 */
  UINT8                    PcdeMMCBootMode;           /* Offset 0x0026 */
  UINT8                    PcdEnableSdio;             /* Offset 0x0027 */
  UINT8                    PcdEnableSdcard;           /* Offset 0x0028 */
  UINT8                    PcdEnableHsuart0;          /* Offset 0x0029 */
  UINT8                    PcdEnableHsuart1;          /* Offset 0x002A */
  UINT8                    PcdEnableSpi;              /* Offset 0x002B */
  UINT8                    ReservedUpdSpace1;         /* Offset 0x002C */
  UINT8                    PcdEnableSata;             /* Offset 0x002D */
  UINT8                    PcdSataMode;               /* Offset 0x002E */
  UINT8                    PcdEnableAzalia;           /* Offset 0x002F */
  UINT32                   AzaliaConfigPtr;           /* Offset 0x0030 */
  UINT8                    PcdEnableXhci;             /* Offset 0x0034 */
  UINT8                    PcdEnableLpe;              /* Offset 0x0035 */
  UINT8                    PcdLpssSioEnablePciMode;   /* Offset 0x0036 */
  UINT8                    PcdEnableDma0;             /* Offset 0x0037 */
  UINT8                    PcdEnableDma1;             /* Offset 0x0038 */
  UINT8                    PcdEnableI2C0;             /* Offset 0x0039 */
  UINT8                    PcdEnableI2C1;             /* Offset 0x003A */
  UINT8                    PcdEnableI2C2;             /* Offset 0x003B */
  UINT8                    PcdEnableI2C3;             /* Offset 0x003C */
  UINT8                    PcdEnableI2C4;             /* Offset 0x003D */
  UINT8                    PcdEnableI2C5;             /* Offset 0x003E */
  UINT8                    PcdEnableI2C6;             /* Offset 0x003F */
  UINT8                    PcdEnablePwm0;             /* Offset 0x0040 */
  UINT8                    PcdEnablePwm1;             /* Offset 0x0041 */
  UINT8                    PcdEnableHsi;              /* Offset 0x0042 */
  UINT8                    PcdIgdDvmt50PreAlloc;      /* Offset 0x0043 */
  UINT8                    PcdApertureSize;           /* Offset 0x0044 */
  UINT8                    PcdGttSize;                /* Offset 0x0045 */
  UINT8                    ReservedUpdSpace2[5];      /* Offset 0x0046 */
  UINT8                    PcdMrcDebugMsg;            /* Offset 0x004B */
  UINT8                    ISPEnable;                 /* Offset 0x004C */
  UINT8                    PcdSccEnablePciMode;       /* Offset 0x004D */
  UINT8                    IgdRenderStandby;          /* Offset 0x004E */
  UINT8                    TxeUmaEnable;              /* Offset 0x004F */
  UINT8                    PcdOsSelection;            /* Offset 0x0050 */
  UINT8                       eMMC45DDR50Enabled;            /* Offset 0x0051 */
  UINT8                       eMMC45HS200Enabled;            /* Offset 0x0052 */
  UINT8                       eMMC45RetuneTimerValue;        /* Offset 0x0053 */
  UINT8                       PcdEnableIgd;                  /* Offset 0x0054 */
  UINT8                       UnusedUpdSpace1[155];          /* Offset 0x0055 */
  MEMORY_DOWN_DATA            PcdMemoryParameters;           /* Offset 0x00F0 */
  UINT16                      PcdRegionTerminator;           /* Offset 0x0100 */
} UPD_DATA_REGION;

#define VPD_IMAGE_ID    0x3157454956594C56        /* 'VLYVIEW1' */
#define VPD_IMAGE_REV   0x00000304

typedef struct _VPD_DATA_REGION {
  UINT64                      PcdVpdRegionSign;              /* Offset 0x0000 */
  UINT32                      PcdImageRevision;              /* Offset 0x0008 */
  UINT32                      PcdUpdRegionOffset;            /* Offset 0x000C */
  UINT8                       UnusedVpdSpace0[16];           /* Offset 0x0010 */
  UINT32                      PcdFspReservedMemoryLength;    /* Offset 0x0020 */
  UINT8                       PcdPlatformType;               /* Offset 0x0024 */
  UINT8                       PcdEnableSecureBoot;           /* Offset 0x0025 */
} VPD_DATA_REGION;

#pragma pack()
```

PcdPlatformType - This is a static configuration option and can be set only through the Binary Configuration Tool. The valid options for this configuration item are Non-ECC Memory Platform and ECC Memory Platform. The default for this configuration option is Non-ECC Memory Platform.

PcdLpssSioEnablePciMode - This configuration option selects the operating mode for the Serial IO controllers like I2C and HSUART in the Intel® Atom™ Processor E3800 Product Family. These controllers can operate in (i) PCI mode, in which they will be visible to the OS through their PCI configuration space and in (ii) ACPI mode, in which they will be visible to the OS as ACPI devices. In ACPI configuration, to avoid conflict in resource windows, the boot loader should allocate the resources for these SIO controllers from a separate resource pool which is not covered by the ACPI PCI Host Bridge's resources as reported in the _CRS method.

9 Tools

A Binary Configuration Tool (BCT) will be 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 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 supports secure boot mechanism as supported by the Intel® Atom™ Processor E3800 Product Family. Please refer to the *Secure Boot Implementation in the Sample Boot Loader using Intel® FSP for Intel® Atom™ Processor E3800 Product Family Platforms README* (#528703) for details.

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 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 information 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_entry_func ()
{
  volatile register UINT8 *ptr asm ("eax");

  //
  // This label will be called directly using a
  // ROM based stack since the temporary memory is not available
  // yet. In order to make it work, C function prologue has to be
  // skipped and only register variable can be used.
  // EAX needs to point to the FSP base address before the call.
  // On return, the EAX will point to the FSP INFO header if
  // successful.If mulitple FVs are used for the FSP, the EAX
  // should point to the FV base address and this function will
  // try to locate the FSP INFO header inside this FV. On return,
  // the EAX will point to the FSP INFO header inside the boot FV
  // if the header is found and valid. Otherwise, NULL will be
  // returned in EAX.
  //
  __asm__ __volatile__ (
      ".global find_fsp_entry\n\t"
      "find_fsp_entry:\n\t"
    );

  //
  // 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 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 ( (((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.

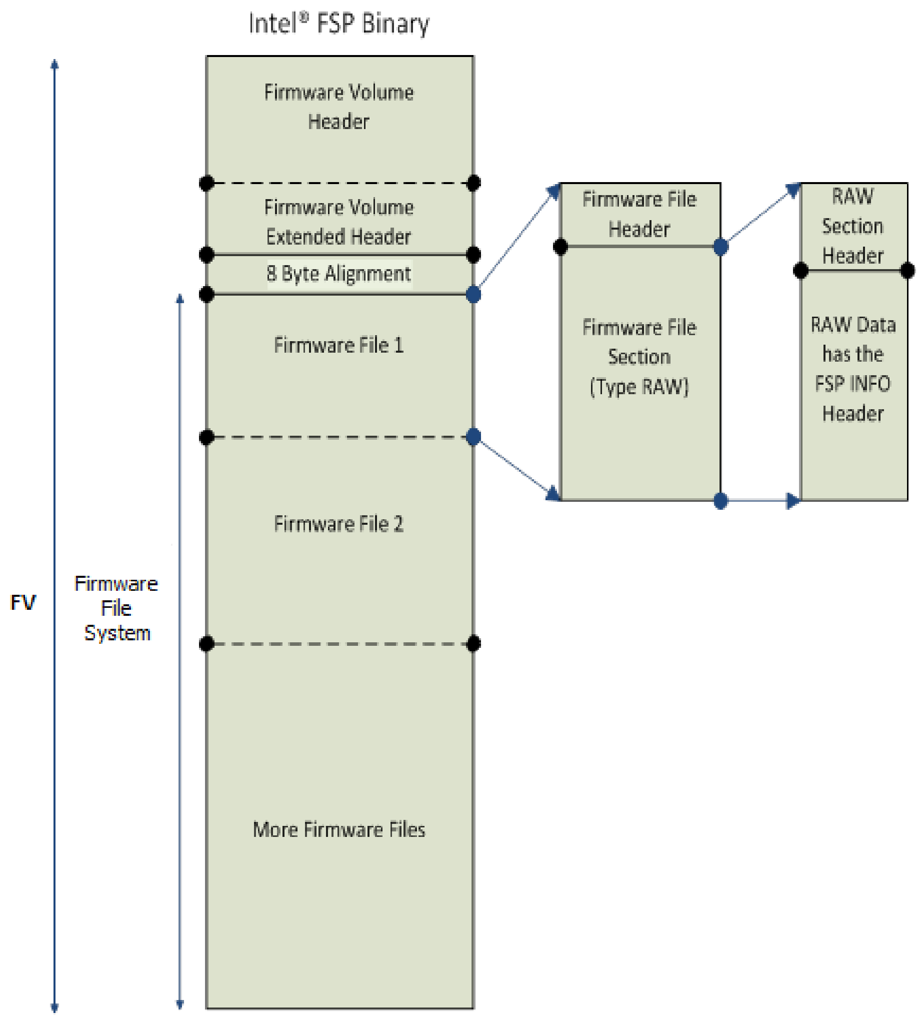
```
  lea    findFspHeaderStack, %esp
  jmp    find_fsp_entry

 findFspHeaderStack:
  .align 4
  .long  findFspHeaderDone

 findFspHeaderDone:
```

A pictorial representation of the data structures that we parse in the above code is given in Figure 2. Data Structures

Figure 2. Data Structures


Intel® FSP Binary
Firmware Volume Header
Firmware Volume Extended Header
8 Byte Alignment
Firmware File 1
Firmware File 2
More Firmware Files
FV
Firmware File System
Firmware File Header
Firmware File Section (Type RAW)
RAW Section Header
RAW Data has the FSP INFO Header