intel

# EFI 1.10 Sample Implementation
# SAL to EFI Handoff State

## Itanium Processor Family Based Platforms

***Draft for Review***

---

Version 1.10

May 7, 2002

# Revision History

| Revision | Revision History | Date | Author |
|---|---|---|---|
| 1.10 | Initial review draft | 1/31/02 | Intel |

intel

# Contents



## Tables



## Figures

# 1 Introduction

This document describes the entry point to EFI in the EFI 1.10 Sample Implementation along with its associated handoff state. Only the EFI entry point for Itanium Processor Family platforms is covered, and the design described here is one of many possible implementations for an EFI compliant platform. The design described here applies to a platform that initializes its SAL environment first, and then passes control to the EFI environment from the EFI 1.10 Sample Implementation.

## 1.1 Organization of this Document

This specification is organized as follows:

**Table 1-1. Specification Organization and Contents**

| Chapter | Description |
|---|---|
| Chapter 1: Introduction | Introduction. |
| Chapter 2 : Overview | Provides an overview of the SAL to EFI Transition. |
| Chapter 3: EFI Main Entry | Definition of the main entry point to EFI from the SAL. This is the function that is passed the SAL to EFI Handoff State. |

## 1.2 Goals

This document describes the entry point to EFI along with the handoff state from the SAL. Given this information, and OEM or IBV should be able to combine a customized SAL implementation with the EFI 1.10 Sample Implementation for an Itanium Processor Family platform.

## 1.3 Target Audience

This document is intended for the following readers:

- OEMs who will be creating Intel Architecture-based platforms intended to boot shrink-wrap operating systems.
- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in Intel Architecture-based products.

## 1.4 Related Information

The following publications and sources of information may be useful to you or are referred to by this specification:

- *Extensible Firmware Interface Specification*, Version 1.10, Intel Corporation, 2001.

- *PCI 2.2 Specification.*

## 1.5 Conventions Used in This Document

This document uses typographic and illustrative conventions described below.

### 1.5.1 Data Structure Illustrations

The Intel Architecture processors of the IA-32 family are “little endian” machines. This means that the low-order byte of a multi-byte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the IA-64 family may be configured for both “little endian” and “big endian” operation.

For the purposes of this specification, illustrations of data structures in memory will always show the lowest addresses at the bottom and the highest addresses at the top of the illustration, as shown in Figure 1-2. Bit positions are numbered from right to left.

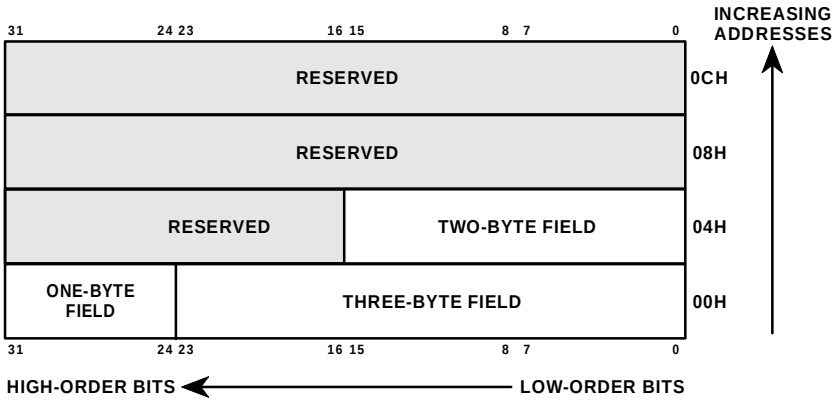


**Figure 1-2. Memory Layout Conventions**

In some memory layout descriptions, certain fields are marked **RESERVED**. Software should initialize these fields as binary zeros, but should otherwise treat them as having a future, though unknown effect. Software should avoid any dependence on the values in the reserved fields.

### 1.5.2 Typographic Conventions

The following typographic conventions are used in this document to illustrate programming concepts:

| | |
|---|---|
| **`Prototype`** | This typeface is use to indicate prototype code. |
| *`Argument`* | This typeface is used to indicate arguments. |
| `Name` | This typeface is used to indicate actual code or a programming construct. |
| **register** | This typeface is used to indicate a processor register |

# 2 Overview

## 2.1 SAL to EFI Transition

The transition from the end of the SAL initialization to the beginning of the EFI initialization occurs in two phases. The first phase initializes the EFI Memory Services and allocates memory for the stacks and BSP store areas that will be used for the second phase of EFI initialization. The SAL is responsible for switching to these new stacks. The second phase completes the initialization of the EFI services, and hands control to the EFI Boot Manager. During the second phase of EFI initialization, several tables are added to the list of configuration tables that are maintained through the EFI System Table. These include the SAL System Table, MPS Table, ACPI Table, and the PCI Option ROM Table. The following flow diagram illustrates the two phases of the SAL to EFI transition.

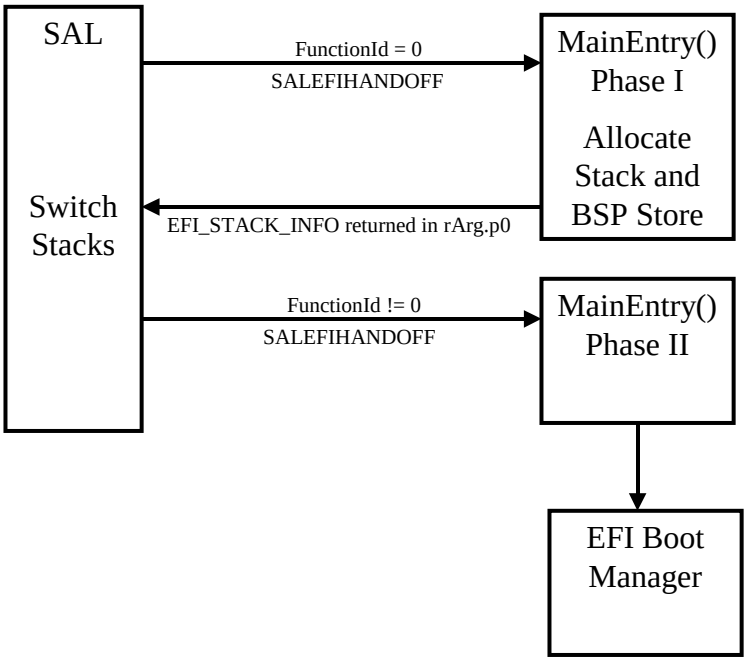


**Figure 2-1. SAL to EFI Transition**

The following diagram shows a possible state of the EFI System Table after all the configuration table entries have been added. Each of the configuration table entries can be added with the EFI Boot Service **`InstallConfigurationTable()`**.

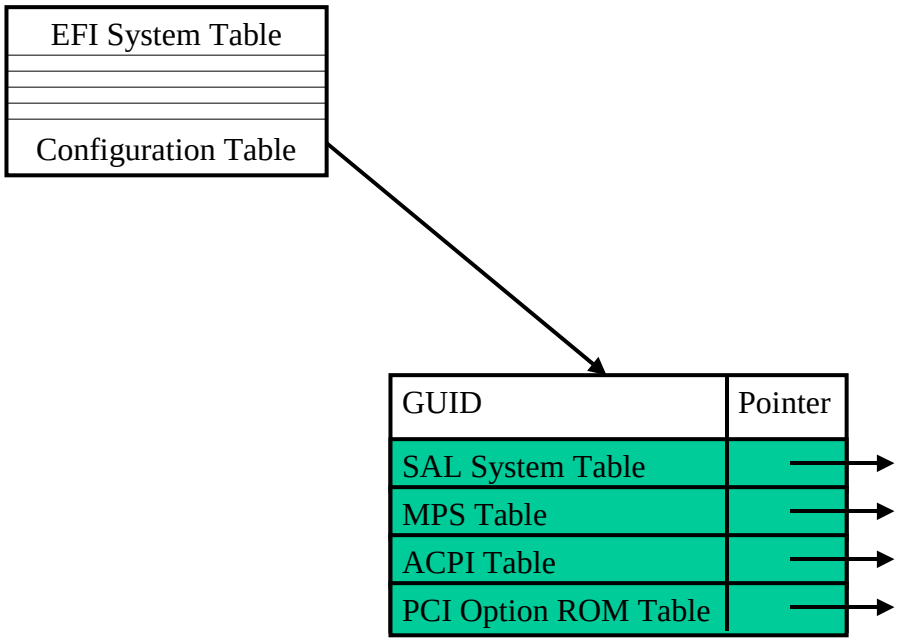


**Figure 2-2. Configuration Table constructed in MainEntry()**

Adding the SAL System Table, MPS Table, and ACPI Table is fairly straightforward. These table pointers are collected by the SAL and are passed to EFI through the SALEFIHANDOFF data structure. The PCI Option ROM Table is more interesting. Since the EFI 1.10 Sample Implementation does not include a PCI resource manager, it is assumed that the SAL or some other firmware component performed PCI resource allocations for the entire set of PCI devices present in the system. During these PCI resource allocations, if any PCI option ROMs are discovered, the contents of the ROM must be copied to a buffer in memory, and a data structure that describes this PCI option ROM must be constructed. These data structures are combined into an array and are included in the SALEFIHANDOFF data structure. In second phase of EFI initialization, the table of PCI option ROMs is added to the Configuration Table that is part of the EFI System Table. This provides a mechanism for the PCI Bus Driver to obtain the contents of the PCI option ROMs. The PCI Bus Driver will look for the PCI Option ROM entry in the Configuration Table, and will use that information to load and start the EFI 1.10 drivers that are found in those option ROM images.

## 2.2 PCI Option ROM Table Construction

The following figure shows the boot flow from the PCI Resource Manager, to the SAL, to EFI, and to the PCI Bus Driver. An EFI 1.10 driver that is stored in a PCI option ROM must be passed through this sequence for it to get properly loaded and started by the PCI Bus Driver.

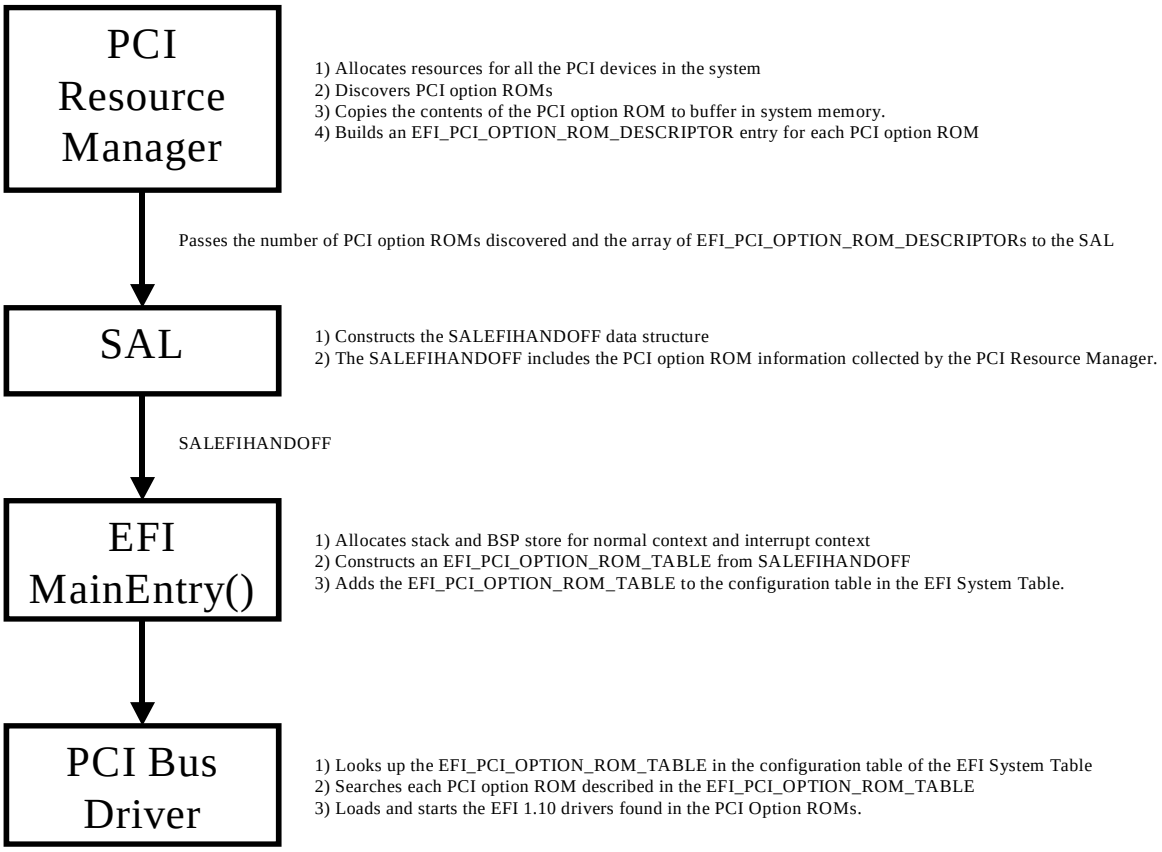


**Figure 2-3. PCI option ROM Table Boot Flow**

The following chapter describes the entry point of EFI and the data structures used to pass information between the different firmware components shown in Figure 2-3.

# 3
# EFI Main Entry

## 3.1 MainEntry Function

The **MainEntry()** function is the C entry point to EFI. This function is responsible for initializing the EFI environment on a platform, and handing control to the EFI Boot Manager.

```
rArg
MainEntry (
  IN UINTN            FunctionId,
  IN SALEFIHANDOFF    *SalEfiHandoff
  );
```

### Parameters

| | |
|---|---|
| *FunctionId* | Specifies the operation that **MainEntry()** is to perform. The SAL will call **MainEntry()** twice. The first call is with a *FunctionId* value of 0. This call is made by the SAL to request EFI to initialize its memory services, and allocate memory for the stack and BSP for normal and interrupt contexts. The second call is with a non-zero *FunctionId* value. This is the final call from the SAL to EFI, and EFI must initialize the rest of its environment and pass control to the EFI Boot Manager. |
| *SalEfiHandoff* | A pointer to the **SALEFIHANDOFF** data structure. This data structure is defined in Section 3.2. |

### Description

This function initializes the EFI environment on a platform. This initialization it divided into two distinct phases. The first phase is when the SAL calls **MainEntry()** with a *FunctionId* of 0. For this phase, **MainEntry()** is responsible for initializing the EFI Memory Services, and performing four memory allocations. These allocations are for the memory stack that EFI will use for normal context, the BSP store that EFI will use for normal context, the memory stack that EFI will use in interrupt context, and the BSP store that EFI will use in interrupt context. The pointers to these four memory allocations are returned in an **EFI_STACK_INFO** structure (Section 3.6). The pointer to the **EFI_STACK_INFO** structure is returned in the *p0* field of the **rArg** return value (Section 3.5). The SAL will use these memory allocations to switch to the new stack and BSP store for normal context before calling **MainEntry()** the second time.

The second phase is when the SAL calls **MainEntry()** with a non-zero *FunctionId* value. **MainEntry()** will never return to the SAL from this phase of execution. **MainEntry()** is responsible for initializing the rest if the EFI environment, and passing control to the EFI Boot Manager. The function of the EFI Boot Manager is described in the *EFI 1.10 Specification*.

## 3.2 SALEFIHANDOFF Data Structure

A pointer to this data structure is one of the parameters passed to **MainEntry()**. It contains a physical memory map that described where memory is present, and how that memory is being used. It also contains pointers to several industry standard tables including the SAL System Table, the MPS Table, and the ACPI Table. In addition, the NVRAM store for environment variables is described along with the PCI option ROMs that were discovered during PCI resource allocation. It also describes the base address and length of the image that contains **MainEntry()**.

### Data Structure

```
typedef struct {
  UINT64                          MemDescCount;
  EFI_MEMORY_DESCRIPTOR           *MemDesc;
  VOID                            *SALPROC_Entry;
  VOID                            *SalSystemTable;
  VOID                            *MpsTable;
  VOID                            *AcpiTable;
  VOID                            *SALCallBack;
  UINT64                          EFIGPValue;
  UINT64                          NVRAMBanks;
  UINT64                          NVRAMSize;
  UINT64                          PciOptionRomCount;
  EFI_PCI_OPTION_ROM_DESCRIPTOR   *PciOptionRomDescriptors;
  VOID                            *EfiCoreBaseAddress;
  UINT64                          *EfiCoreLength;
} SALEFIHANDOFF;
```

*MemDescCount* — Specifies the number of memory descriptors in the *MemDesc* array.

*MemDesc* — A pointer to an array of EFI memory descriptors. *MemDescCount* specifies the number of entries in the array. Type **EFI_MEMORY_DESCRIPTOR** is defined in the *EFI 1.10 Specification*. This array describes all the physical memory available in the system, and how all that memory is currently being used. This information is used to initialize the EFI memory services like **AllocatePool()**, **FreePool()**, **AllocatePages()**, **FreePages()**, and **GetMemoryMap()**.

*SALPROC_Entry* — Reserved. Must be 0.

*SalSystemTable* — A pointer to the SAL System Table. **MainEntry()** is responsible for registering the SAL System Table with the EFI System Table using the EFI Boot Service **InstallConfigurationTable()**. If

| | |
|---|---|
| | *SalSystemTable* is **NULL**, then the table should not be registered. |
| *MpsTable* | A pointer to the MPS Table. **MainEntry()** is responsible for registering the MPS Table with the EFI System Table using the EFI Boot Service **InstallConfigurationTable()**. If *MpsTable* is **NULL**, then the table should not be registered. |
| *AcpiTable* | A pointer to the ACPI 1.0 Table. **MainEntry()** is responsible for registering the ACPI 1.0 Table with the EFI System Table using the EFI Boot Service **InstallConfigurationTable()**. If *AcpiTable* is **NULL**, then the table should not be registered. |
| *SALCallBack* | Entry point used to make private calls from EFI back into the SAL. These calls are platform specific services that are required by EFI. |
| *EFIGPValue* | The 64 bit global pointer (GP) register value for EFI. The SAL is responsible to loading and starting EFI, so this is the GP value that the SAL collected when it loaded EFI. |
| *NVRAMBanks* | The number if banks of NVRAM storage that are used to store EFI environment variables. |
| *NVRAMSize* | The size of each NVRAM storage bank used to store EFI environment variables. |
| *PciOptionRomCount* | Specifies the number of PCI option ROMs that are described in *PciOptionRomDescriptors*. |
| *PciOptionRomDescriptors* | A pointer to the array of PCI option ROM descriptors. *PciOptionRomCount* specifies the number of entries in the array. Type **EFI_PCI_OPTION_ROM_DESCRIPTOR** is defined in Section 3.4. This array describes all the PCI option ROMs that were discovered during PCI resource allocation. This array is used by the PCI Bus Driver to load and execute EFI 1.10 drivers that are stored in PCI option ROMs. |
| *EfiCoreBaseAddress* | A pointer to the base address of the EFI core image that was loaded by the SAL and contains the entry point **MainEntry()**. |
| *EfiCoreLength* | Specifies the length, in bytes, of the EFI core image that was loaded by the SAL and contains the entry point **MainEntry()**. |

## 3.3 EFI_PCI_OPTION_ROM_TABLE Data Structure

This data structure is a configuration table that is added to the EFI System Table in **MainEntry()**. It describes the PCI Option ROMs that were discovered during PCI resource allocation and were passed to **MainEntry()** through the **SALEFIHANDOFF** structure.

### GUID

```
#define EFI_PCI_OPTION_ROM_TABLE_GUID \
  {0x7462660f,0x1cbd,0x48da,0xad,0x11,0x91,0x71,0x79,0x13,0x83,0x1c}
```

### Data Structure

```
typedef struct {
  UINT64                          PciOptionRomCount;
  EFI_PCI_OPTION_ROM_DESCRIPTOR   *PciOptionRomDescriptors;
} EFI_PCI_OPTION_ROM_TABLE;
```

*PciOptionRomCount* — Specifies the number of PCI option ROMs that are described in *PciOptionRomDescriptors*.

*PciOptionRomDescriptors* — A pointer to the array of PCI option ROM descriptors. *PciOptionRomCount* specifies the number of entries in the array. Type **EFI_PCI_OPTION_ROM_DESCRIPTOR** is defined in Section 3.4. This array describes all the PCI option ROMs that were discovered during PCI resource allocation. This array is used by the PCI Bus Driver to load and execute EFI 1.10 drivers that are stored in PCI option ROMs.

## 3.4 EFI_PCI_OPTION_ROM_DESCRIPTOR Data Structure

The *PciOptionRomDescritors* field of the **SALEFIHANDOFF** contains a pointer to this data structure type.

### Data Structure

```
typedef struct {
  EFI_PHYSICAL_ADDRESS    RomAddress;
  EFI_MEMORY_TYPE         MemoryType;
  UINT32                  RomLength;
  UINT32                  Seg;
  UINT8                   Bus;
  UINT8                   Dev;
  UINT8                   Func;
  BOOLEAN                 ExecutedLegacyBiosImage;
  BOOLEAN                 DontLoadEfiRom;
} EFI_PCI_OPTION_ROM_DESCRIPTOR;
```

*RomAddress* — The base address of the in memory copy of the PCI option ROM image. Type **EFI_PHYSICAL_ADDRESS** is defined in the *EFI 1.10 Specification.*

*MemoryType* — The type of memory that the in memory copy of the PCI option ROM is consuming. Type **EFI_MEMORY_TYPE** is defined in the *EFI 1.10 Specification.*

*RomLength* — The length of the PCI option ROM image in bytes. The length of a PCI option ROM can be found by walking the list of images in the PCI option ROM. The method for walking this liked list of images and determining the size of each image is defined in the *PCI 2.2 Specification.*

*Seg* — The PCI segment number of the PCI controller with which this PCI option ROM is associated.

*Bus* — The PCI bus number of the PCI controller with which this PCI option ROM is associated. The valid range for this field is 0..255.

*Dev* — The PCI device number of the PCI controller with which this PCI option ROM is associated. The valid range for this field is 0..31.

*Func* — The PCI function number of the PCI controller with which this PCI option ROM is associated. The valid range for this field is 0..7.

| | |
|---|---|
| *ExecutedLegacyBiosImage* | If this field is **TRUE**, then a legacy IA-32 option ROM image was found in this option ROM, and that legacy IA-32 image was executed. If this field is **FALSE**, then a legacy IA-32 option ROM image was not found in this PCI option ROM. |
| *DontLoadEfiRom* | If this field is **TRUE**, then none of the EFI 1.10 drivers contained in this PCI option ROM should be loaded and executed by the PCI Bus Driver. If this field is **FALSE**, then all the EFI 1.10 drivers contained in this PCI option ROM should be loaded and executed by the PCI Bus Driver. |

## 3.5 rArg Data Structure

A pointer to this data structure is the return value from **`MainEntry()`**.

### Data Structure

```
typedef struct {
  UINT64     p0;
  UINT64     p1;
  UINT64     p2;
  UINT64     p3;
} rArg;
```

*p0* A 64 bit value that is returned to the SAL in register **R8**.

*p1* A 64 bit value that is returned to the SAL in register **R9**.

*p2* A 64 bit value that is returned to the SAL in register **R10**.

*p3* A 64 bit value that is returned to the SAL in register **R11**.

## 3.6 EFI_STACK_INFO Data Structure

This data structure is returned in the *p0* field of the **rArg** structure that is returned from **MainEntry()** when *FunctionId* is 0.

### Data Structure

```
typedef struct {
  UINT64    EfiStackAddr;
  UINT64    EfiBspAddr;
  UINT64    EfiIntStackAddr;
  UINT64    EfiIntBspAddr;
} EFI_STACK_INFO;
```

| | |
|---|---|
| *EfiStackAddr* | The base address of the stack to use in the normal EFI context. |
| *EfiBspAddr* | The base address of the BSP store to use in the normal EFI context. |
| *EfiIntStackAddr* | The base address of the stack to use in interrupt context. |
| *EfiIntBspAddr* | The base address of the BSP store to use in interrupt context. |