Intel® Platform Innovation Framework for EFI

Compatibility Support Module Specification

Revision 0.97

September 4, 2007

Intel order number: D56958-001

Revision History

Revision	Revision History	Date
0.9	• First public release.	9/16/03
0.91	• Added PciExpressBase parameter to EFI_COMPATIBILITY16_TABLE. • Renamed GetOemInt15Data to GetOemIntData and expanded it to support any software INT. • Modified PrepareToBootEfi to return BBS table. BBS Table updated to return AssignedDriveNumber. • Added GetTpmBinary. • Combined several LegacyBiosPlatform APIs into three APIs. Updated BBS_TABLE and EFI_COMPATIBILITY16_TABLE.	2/01/05
0.95	• Added following modes to EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.GetPlatformInfo() • EfiGetPlatformPciPmmSize • EfiGetPlatformEndRomShadowaddr • SMM_ENTRY clarification added 3/01/05 • Re-added elements to EFI_COMPATIBILITY16_TABLE that inadvertently go dropped.	4/22/05
0.96	• Modified EFI_IA32_REGISTER_SET structure definition to support 32bit register across Thunk interface • Modified some field definitions in EFI_COMPATIBLITY16_TABLE and EFI_COMPATIBILITY16_BOOT_TABLE structures to ensure that the space occupied by these structures in IA32 and x64 architecture is identical. • Reintroduced the BiosLessThan1MB field in EFI_TO_COMPATIBILITY16_INIT_TABLE to maintain compatibility with previous versions of the specification. Added the LowPmmMemory and LowPmmMemorySizeInBytes fields at the end of EFI_TO_COMPATIBILITY16_INIT_TABLE. • Modified EFI_WORD_REGS, EFI_DWORD_REGS, and EFI_EFLAGS_REG structure definitions to make these structures identical to those used in Legacy Soft SMI so that same structures can be used in Thunk as well as in Legacy Soft SMI. • Changed the output parameter definition in EFI_LEGACY_BIOS_PROTOCOL.GetLegacyRegion() to match the implemented code. • Added two more allowed values HDD_MASTER_ATAPI_ZIPDISK and HDD_SLAVE_ATAPI_ZIPDISK in the Status field of HDD_INFO to include support for ZIP disk. • Editing and formatting pass.	4/18/06

Revision	Revision History	Date
0.97	• Added OpromDestinationSegment field to the EFI_DISPATCH_OPROM_TABLE structure to indicate where the OpROM may have been relocated to.	9/04/07

Table of Contents

Tables

Figures

1 Introduction

1.1 Overview

This specification describes the high-level design of the Compatibility Support Module (CSM) code that is required for an implementation of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). The CSM provides compatibility support between the Framework and traditional, legacy BIOS code and allows booting a traditional OS or booting an EFI OS off a device that requires a traditional option ROM (OpROM). This specification does the following:

- Describes the basic components of the CSM
- Defines how to use traditional BIOS option ROMs (OpROMs) to boot an EFI operating system (OS)
- Defines how to boot a traditional OS
- Provides code definitions for compatibility-related services, protocols, functions, and type definitions that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*

1.2 Scope

This document describes the high-level design of the Compatibility Support Module (CSM) code developed for the Intel® Platform Innovation Framework for EFI (hereafter referred to as "the Framework"). This document will define how to do the following:

- Use traditional BIOS option ROMs (OpROMs) to boot an EFI Operating System (OS)
- How to boot a traditional OS

This document's primary focus is how to bind EFI and compatibility support code together. It does not define internal details of the CSM design

1.3 Rationale

There is always a transitional period between the introduction of a new technology and the technology that it replaces. The addition of compatibility support code to EFI bridges this transitional period. The compatibility support code allows booting a traditional OS or booting an EFI OS off a device that requires a traditional OpROM.

2
Design Discussion

2.1 Definitions of Terms

The following definitions, except where noted, are not EFI specific. See the master Framework glossary for definitions of other Framework terms; see "Typographic Conventions" later in this section for the URL.

16-bit legacy

The traditional PC environment and includes traditional OpROMs and Compatibility16 code.

Compatibility16

The traditional BIOS with POST and BIOS Setup removed. Executes in 16-bit real mode.

CompatibilitySmm

Any IBV-provided SMM code to perform traditional functions that are not provided by EFI.

CSM

Compatibility Support Module. The combination of EfiCompatibility, CompatibilitySmm, and Compatibility16.

EfiCompatibility

32-bit EFI code to generate data for traditional BIOS interfaces or EFI Compatibility Support Module drivers, or code to invoke traditional BIOS services.

IBV

Independent BIOS vendor.

NV

Nonvolatile.

OpROM

Option ROM.

PIC

Programmable Interrupt Controller.

PMM

Post Memory Manager.

reverse thunk

The code to transition from 16-bit real mode to native execution mode and back.

SMM

System Management Mode.

thunk

The code to transition from native execution mode to 16-bit real mode and back.

traditional OpROM

16-bit OpROMs that are executed in real mode.

2.2 CSM-Specific References

The following reference is useful for implementing CSM code. See References in the master help system for additional related specifications.

- IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference. Second edition. IBM Corporation, IBM No. S68X-2341-00, 1988.

2.3 CSM Overview

2.3.1 Legacy Overview

This document describes the additional EfiCompatibility functionality (over the standard EFI) that is provided to support traditional BIOS (non-EFI) OSs and/or traditional OpROMs. This functionality along with the associated IBV Compatibility16 and CompatibilitySmm code is called the *Compatibility Support Module (CSM).*

It is expected that traditional OpROM support will be required longer than traditional OS support. The figure below presents a block-diagram-level overview of how a legacy system operates using the CSM.

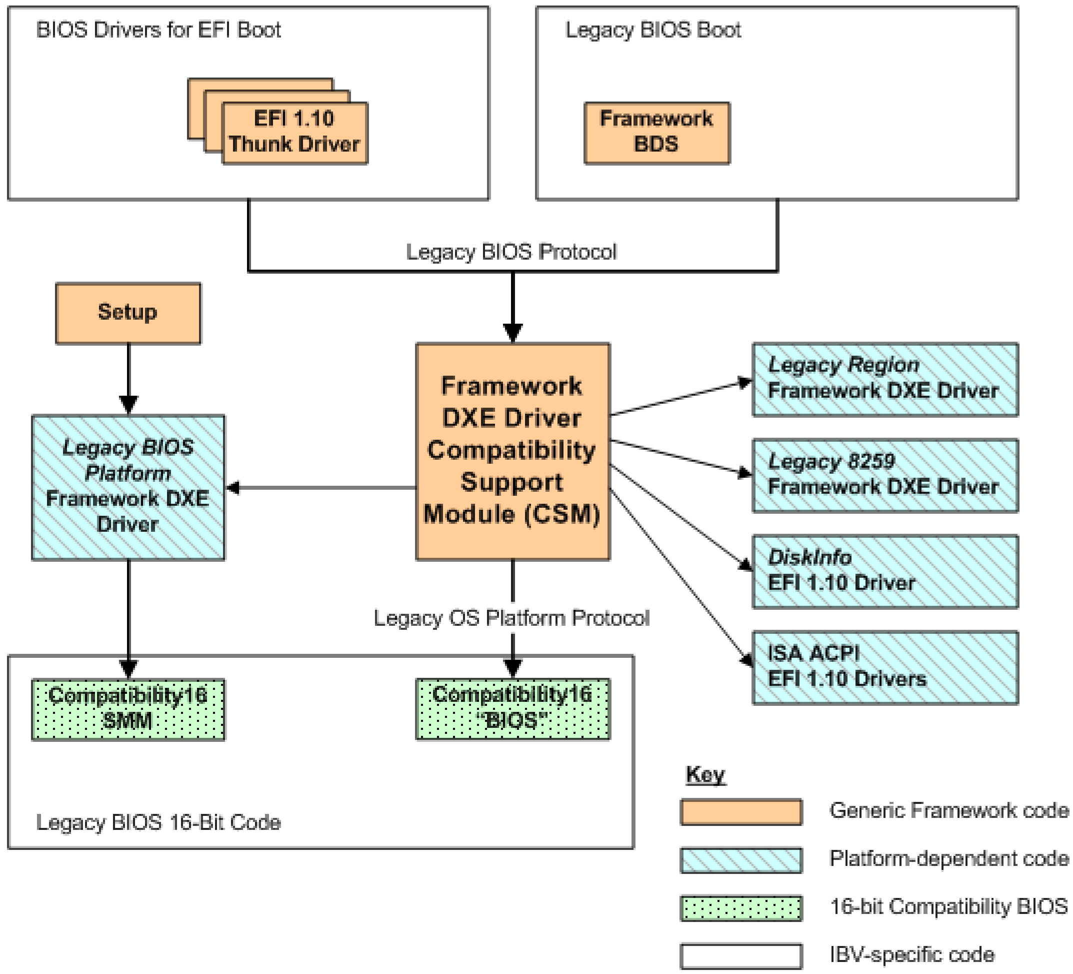

Figure 1 Compatibility Overview

2.3.2 Differences between Traditional BIOS and EFI

An EFI system differs from the traditional BIOS POST in that only minimal system configuration takes place until the Boot Device Selection (BDS) phase (equivalent to traditional POST INT19). Video is not required until the BDS phase, nor are other OpROMs dispatched until the BDS phase. Likewise, BIOS Setup is entered from the BDS phase. These differences place the policy to invocate (or not invocate) the CSM in BDS and make this policy an integral part of BDS.

The policy is predominately set by the following three classes of information: OS being booted, boot drive selection, and Device OpROM selection.

2.3.2.1 OS Being Booted

The selection of booting a traditional (non-EFI-aware) OS dictates that any OpROM being dispatched must be a traditional OpROM rather than an EFI OpROM. This requirement means that the CSM code must be activated and invoked.

2.3.2.2 Boot Device Selection

A boot device that has only a traditional OpROM associated with it requires the CSM code to be activated and invoked.

2.3.2.3 Device OpROM Selection

A BDS policy to initialize all devices might require the CSM code to be activated and invoked when a non-boot device has only a traditional OpROM associated with it.

2.3.3 BDS Legacy Flow

The figure below is a flowchart showing the decisions and operations that take place during BDS in a legacy environment.

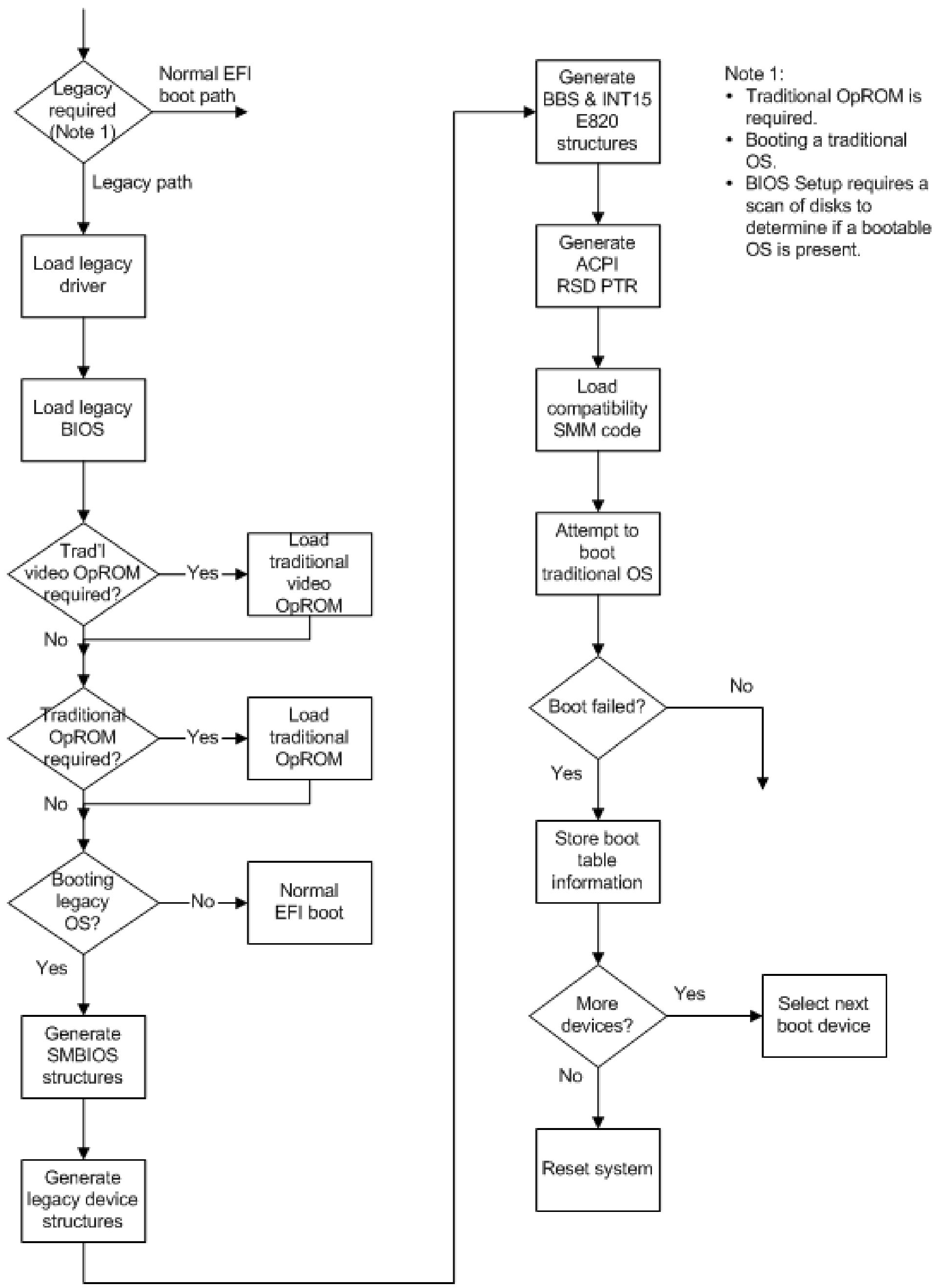

Figure 2 BDS Legacy Flow

2.3.4 Components of CSM

The CSM code consists of the following six main components or functional areas:

EFI Compatibility Support Module initialization code

A set of modules to initialize the CSM data structures and the traditional Post Memory Manager (PMM).

Dispatching traditional OpROMs:

A set of EfiCompatibility drivers to simulate traditional INTs, and code to place traditional OpROMs within the traditional OpROM memory region and invoke the OpROMs.

Translating traditional devices into CSM structures and updating traditional tables:

A set of Compatibility16 functions to extract data from EFI and then convert the extracted data into standard Compatibility16 data structures.

Booting the OS:

A set of EfiCompatibility procedures to boot a traditional OS or boot an EFI-aware OS off a device controller by using a traditional OpROM.

Thunk and reverse thunk code:

There are two flavors of the thunk code; Far call and Interrupt (INT). Both of these internally are assembly-based calls rather than C-based calls but externally are C-based. The *thunk* provides a mechanism to transfer control from EFI to 16-bit traditional code and return to EFI after completion. The *reverse thunk* provides a mechanism to transfer from 16-bit traditional code to 32-bit EFI code and return after completion. It is expected that the reverse thunk will be invoked only in exceptional conditions.

Runtime traditional code provided by an IBV:

Consists of the following:

The Compatibility16 runtime code

Traditional SMM code residing with EFI SMM code

EfiCompatibility platform protocol internals

2.4 CSM Architecture

2.4.1 Overview

The CSM provides additional functionality to EFI. This additional functionality permits the loading of a traditional OS or the use of a traditional OpROM. This new functionality requires the following:

- New 32-bit code that operates in the EFI environment (EfiCompatibility)
- A stripped-down traditional 16-bit real-mode BIOS (Compatibility16)
- Code to transition between the 32-bit and 16-bit code (thunk and reverse thunk)
- Optionally code in SMM (CompatibilitySmm) to perform traditional functions not taken over by EFI SMM

Outside this additional functionality are the traditional OpROMs, regardless if they reside onboard or offboard.

Several pieces of code make up the CSM, as listed in the table below.

Table 1 Components of CSM

Component	Description
EfiCompatibility	32-bit code that interfaces with EFI EfiCompatibility is comprised of several EFI drivers. These drivers fall into the following four categories: • Drivers that are platform and hardware neutral. They provide the foundation of the CSM and do not change from platform to platform. • Drivers that are platform and chipset neutral They control traditional hardware such as the 8259 PIC. • Drivers that are chipset dependant. They control chipset hardware that changes from platform to platform. An example is the code to perform PCI IRQ steering. • Drivers that are platform specific.
Compatibility16	Stripped-down, traditional IBV, 16-bit real-mode code consisting mainly of the traditional software INT runtime support.
CompatibilitySmm	Optional code in SMM to perform traditional functions that are not taken over by the EFI SMM.
Thunk and Reverse Thunk	Thunk code is used to switch from EfiCompatibility (32-bit) to Compatibility16 or traditional OpROMs (16-bit). Reverse thunk code is used to switch from Compatibility16 code or traditional OpROMs (16-bit) to EfiCompatibility (32-bit).

Outside the CSM, but still part of a traditional system, are the traditional 16-bit real-mode OpROMs.

The CSM operates in two distinct environments:

- Booting a traditional or non-EFI-aware OS.
- Loading an EFI-aware OS a device that is controlled by a traditional OpROM.

The first operation, booting a traditional or non-EFI-aware OS, is the traditional environment.

It is expected that traditional OpROMs will be around long after traditional OSs have been replaced by EFI-aware OSs. The code that is required to load an EFI-aware OS is a subset of the code that is required to boot a traditional (non-EFI-aware) OS. The Framework architecture reflects this split to allow removing unneeded code in the future.

See Steps in CSM Driver Interactions for additional details on how the various CSM components interconnect with each other and how external drivers interconnect with the CSM.

2.4.2 EfiCompatibility

The EfiCompatibility consists of the protocols listed in the table below. This EfiCompatibility code is written in C and exists in the EFI environment.

Table 2 EfiCompatibility Protocols

Protocol	Description
Legacy BIOS Protocoll	The primary protocol of the CSM. This protocol is platform and hardware neutral.
Legacy BIOS Platform Protocol	Provides the information that makes this platform unique compared to another platform using the same chipset. This protocol is platform specific.
Legacy Region Protocol	Manages the hardware that allows the region from physical address 0xC0000 to 0xFFFFF to be made read only or read-write. It can optionally manage the hardware that prevents a write in the above region propagating to any aliased memory regions This protocol is chipset specific.
Legacy 8259 Protocol	Manages the hardware controlling the 8259 PIC in 32-bit protected mode and in 16-bit real mode. It keeps track of the interrupt masks and edge/level programming for both modes. This protocol is platform and chipset neutral.
Legacy Interrupt Protocol	Manages the hardware for assigning IRQs to PCI. EFI is polled rather than interrupt driven. This protocol is chipset specific.

Each of the above protocols is described in more detail in the following sections. See EfiCompatibility Code in Code Definitions for the definitions of these protocols.

2.4.2.1 Legacy BIOS Protocol

2.4.2.1.1 LegacyBios Module

The LegacyBios module constitutes the CSM skeleton. The other CSM modules are support modules. The LegacyBios initialization code consists of the following elements:

- Code to initialize itself and to load the Compatibility16 code
- Code to determine the boot device
- Code to load other EfiCompatibility drivers
- Code to load the thunk and reverse thunk code
- Code to interface with Compatibility16 functions
- Code to load and invoke traditional OpROMs, including code to find baseboard traditional OpROMs and to load them
- Code to load a traditional OS

The BDS code, as part of its normal functions, binds a device with a traditional OpROM, but it uses the LegacyBios code to dispatch and initialize the traditional OpROM. The BDS code also generates the various boot device paths, but it uses the LegacyBios code to boot to a traditional OS. The LegacyBios code that is used to boot a traditional OS performs the following actions:

- Extracts EFI data into standard EfiCompatibility structures
- Updates standard BDS, EBDA, and CMOS locations
- Updates hardware with traditional resources (IRQs)

The LegacyBios code parses the data hub or invokes EFI APIs to gather data that is required by Compatibility16 and translates it into a series of standard Compatibility16 data structures. That data is used to update standard traditional data values in the BDA, EBDA, and CMOS. The data is also used by Legacy BIOS Protocol APIs to reprogram traditional devices to traditional resources.

Compatibility16 does not configure low-level device hardware and instead leaves that operation to EFI. EFI does not assign IRQs to devices such as serial ports, but Compatibility16 requires them to be configured with the appropriate IRQs. The LegacyBios code must reconfigure any traditional devices that were configured by EFI into a valid Compatibility16 configuration.

It is expected that there will be a light and full version of the Legacy BIOS Protocol. The light version is for environments where the OS is always an EFI-aware OS that might have traditional OpROMS. The full version is for environments where a traditional OS might be invoked.

2.4.2.1.2 Legacy BIOS Protocol

The Legacy BIOS Protocol, along with the initialization of the LegacyBios driver, provides the foundation of the CSM code. The table below lists the functions that are included in the Legacy BIOS Protocol. See **EFI_LEGACY_BIOS_PROTOCOL** in Code Definitions for the definitions of these functions.

Table 3 Functions in Legacy BIOS Protocol

Functions	Description
BootUnconventionalDevice()	Allows the user to boot off of an unconventional device such as a PARTIES partition.
CheckPciRom()	Checks if a device has a traditional OpROM associated with it. It is used to determine valid traditional OS boot devices, or if a traditional OpROM exists for a device that has no EFI OpROM support.
CopyLegacyRegion()	Allows EFI to copy data to the area specified by **GetLegacyRegion()**. It may be invoked multiple times. This function performs boundary checking via information passed into **GetLegacyRegion()**.
FarCall86()	Allows the 32-bit protected-mode code to perform a far call to 16-bit real-mode code. It is analogous to the **Int86()** function, but a far call is patched instead.
GetBbsInfo()	Allows external drivers to access the internal EfiCompatibility BBS data structures. This function is normally used by BIOS Setup.
GetLegacyRegion()	Allows EFI to reserve an area in the 0xE0000 or 0xF0000 block. This function may be invoked only once.
Int86()	Allows the 32-bit protected-mode code to perform a traditional 16-bit real-mode software interrupt. The function invokes the thunk code to switch to 16-bit real mode, patches an INT instruction with the required software interrupt, loads the IA-32 registers from data in the passed-in register file, and issues the software interrupt. Upon completion of the interrupt, it updates the register file and switches back to 32-bit protected mode.
InstallPciRom()	Installs a traditional OpROM in the 0xC0000 to 0xFFFFF region.
LegacyBoot()	Initiates booting from a traditional OS. The majority of the CSM work is done within this function, because the final commitment to boot from a traditional OS has been made and the boot process will destroy EFI code. This function returns to the caller only in the exceptional condition in which a traditional INT19 failed but control was never passed to an OS first-stage loader. If control was ever passed to an OS first-stage loader, then the Compatibility16 code must issue a reset, because memory may have been written over and EFI corrupted.

PrepareToBootEfi()	Allows an external agent to prepare for booting to an EFI-aware OS. It is a subset of actions taken by **LegacyBoot()**. It causes legacy drive numbers to be assigned.
ShadowAllLegacyOproms()	Allows an external agent to force the loading of legacy OpROMs. A side affect of this function is that all EFI drivers are disconnected and must be reconnected for proper EFI functioning.
UpdateKeyboardLedStatus()	Allows the EfiCompatibility code to synchronize the traditional BIOS BDA with the state that EFI has programmed the keyboard LEDs. This function does not touch hardware. The Compatibility16 code is invoked with the state of the LEDs in case any proprietary information needs to be updated.

2.4.2.1.3 GetBbsInfo(), LegacyBoot(), ShadowAllLegacyOproms(), and PrepareToBootEfi()

BDS and BIOS Setup require certain information to intelligently determine boot devices and require different actions to occur depending upon the type of OS being booted.

Determining what boot devices are available

Both BDS and BIOS Setup need to know the complete list of boot devices to present a comprehensive list to the user. It is possible that a device controller by both an EFI and legacy OpROM may report different results. EFI drivers generate a list of boot devices and then the **EFI_LEGACY_BIOS_PROTOCOL.ShadowAllLegacyOproms()** call is issued. The internal CSM BBS table information is updated during the OpROM initializations. The **EFI_LEGACY_BIOS_PROTOCOL.ShadowAllLegacyOproms()** function will disconnect all EFI devices so a reconnect must be performed after the invocation. **EFI_LEGACY_BIOS_PROTOCOL.GetBbsInfo()** returns the list of legacy boot devices that were discovered. Note that at this time no legacy drive numbers 0x8y have been assigned because the Compatibility16 code has not been issued the **EFI_LEGACY_BIOS_PROTOCOL.PrepareToBootEfi()** function.

Determining the boot OS type

Once the list of boot devices is available, the user selects the boot device. If booting to a traditional OS, BDS issues the **EFI_LEGACY_BIOS_PROTOCOL.LegacyBoot()** function. If booting to an EFI-aware OS and any legacy OpROMs have been initialized, then **EFI_LEGACY_BIOS_PROTOCOL.PrepareToBootEfi()** is issued.

2.4.2.2 Legacy BIOS Platform Protocol

The Legacy BIOS Platform Protocol provides the customization of CSM for both platform configuration and for OEM differentiation. The table below lists the functions that are included in the Legacy BIOS Platform Protocol. See **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** in Code Definitions for the definitions of these functions.

Table 4 Functions in the Legacy BIOS Platform Protocol

Function	Description
GetPlatformHandle()	Finds all handles for the requested entity and returns them sorted by priority. Handle[0] is highest priority.

GetPlatformInfo()	Used to return binary objects or various pieces of data..
GetRoutingTable()	Serves two purposes; it is used for PCI PIRQ routing and for $PIR table information.
PlatformHooks()	Any required hook after a CSM operation.
PrepareToBoot()	Allows any final processing to take place before booting a traditional OS.
SmmInit()	Finds any CompatibilitySmm modules that exist in EFI firmware volumes and registers them with the EFI SMM driver. The number of CompatibilitySmm modules can be zero or greater. There is no maximum number.
TranslatePirq()	Translates the PIRQ reported by the PCI device back through the bridges into the equivalent root PIRQ.

2.4.2.3 Legacy Region Protocol

The Legacy Region Protocol controls the read-write attributes for the region 0xC0000 to 0xFFFFF. The table below lists the functions that are included in the Legacy Region Protocol. See **EFI_LEGACY_REGION_PROTOCOL** in Code Definitions for the definitions of these functions.

Table 5 Functions in Legacy Region Protocol

Function	Description
Decode()	Programs the chipset to decode or not decode regions in the 0xC0000 to 0xFFFFF range. The default is to decode entire range.
Lock()	Programs the chipset to lock (write protect) regions in the 0xC0000 to 0xFFFFF range.
BootLock()	Programs the chipset to lock (write protect) regions in the 0xC0000 to 0xFFFFF range and is invoked just prior to booting a traditional OS. In addition, it ensures that a write to the region does not cause a write at any aliased addresses.
Unlock()	Programs the chipset to unlock (read-write) regions in the 0xC0000 to 0xFFFFF range. In addition, it ensures that a write to the region does not cause a write at any aliased addresses.

2.4.2.4 Legacy 8259 Protocol

The Legacy 8259 Protocol controls the programming of the 8259 PIC in both the EFI (32-bit protected) environment and legacy (16-bit real-mode) environment. The table below lists the functions that are included in the Legacy 8259 Protocol. See **EFI_LEGACY_8259_PROTOCOL** in Code Definitions for the definitions of these functions.

Table 6 Functions in Legacy 8259 Protocol

Functions	Description
SetVectorBase()	Sets the vector base for the 8259 PIC. In the EFI environment, the base is set to 0x1A0 (INT68) for master and 0x1C0 (INT70) for slave PIC. In the legacy environment, the base is set to 0x20 (INT08) for master and 0x1C0 (INT70) for slave PIC. The different master PIC address for EFI prevents the overlaying of interrupts and processor exceptions.
GetMask()	Gets the current settings of the master and slave interrupt mask and/or the edge/level register programming. The caller can specify EFI and/or legacy environment.
SetMask()	Sets the current settings of the master and slave interrupt mask and/or the edge/level register programming. The caller can specify EFI and/or legacy environment.
SetMode()	Sets the current mode (EFI or legacy) and settings of the master and slave interrupt mask and/or the edge/level register programming. This function should not be invoked multiple times in the same mode. Use the **SetMask()** function instead.
GetVector()	Translates an IRQ into an INT. For example, IRQ0 is INT68 for the EFI environment and INT08 for the legacy environment.
EnableIrq()	Enables an interrupt in the EFI environment. Non-CSM drivers normally use this function.
DisableIrq()	Disables an interrupt in the EFI environment. Non-CSM drivers normally use this function.
GetInterruptLine()	Returns the IRQ assigned to the specified PCI device.
EndOfInterrupt()	Generates an End of Interrupt (EOI) command for the specified IRQ.

2.4.2.5 Legacy Interrupt Protocol

The Legacy Interrupt Protocol manages the programming of PCI interrupts. The table below lists the functions that are included in the Legacy Interrupt Protocol. See **EFI_LEGACY_INTERRUPT_PROTOCOL** in Code Definitions for the definitions of these functions.

Table 7 Functions in Legacy Interrupt Protocol

Functions	Description
GetNumberPirqs()	Returns the number of PIRQs that the chipset supports.
GetLocation()	Returns the PCI bus location of the chipset. The $PIR table requires this information.
ReadPirq()	Reads the current PIRQ contents for the indicated PIRQ.
WritePirq()	Writes the current PIRQ contents for the indicated PIRQ.

2.4.3 Compatibility16

The Compatibility16 code is a stripped-down version of a traditional BIOS that removes the POST and BIOS Setup code. This stripped-down BIOS consists of the following:

- Runtime code
- INT18 code
- INT19 code
- A small piece of new code to handle the interface between EfiCompatibility code and Compatibility16 code

The design goal is to have the Compatibility16 code be universal for each class of platforms. Examples are desktop, server, and mobile platforms. This goal implies that the Compatibility16 code is chipset and platform neutral. It controls hardware through traditional hardware interfaces and leaves the chipset programming to EFI and/or EfiCompatibility. This design goal maximizes reusability and minimizes code bugs.

2.4.3.1 Communication between EfiCompatibility and Compatibility16

The communication between EfiCompatibility modules and Compatibility16 occurs using the following three mechanisms:

- Compatability16 Table
- Compatability16 Functions
- Compatability16 Function Data Structures

The following sections discuss these mechanisms in more detail. See Compatibility16 Code in Code Definitions for definitions of these functions and structures.

2.4.3.1.1 Compatibility16 Table

There is a new table, **EFI_COMPATIBILITY16_TABLE**, introduced to the traditional legacy runtime BIOS for CSM support. This table is on a 16-byte boundary and has a signature of “$EFI” when read as a DWORD. The Compatibility16 code has a default table generated at build time. The most important fields are the *Compatibility16CallSegment:Offset*. EfiCompatibility uses this address to issue Compatability16 Functions. The appropriate side fills in the other fields during normal CSM operation.

2.4.3.1.2 Compatibility16 Functions

These functions allow the EfiCompatibility code to communicate with the Compatibility16 code and are an addition to the traditional BIOS runtime code.

These functions provide the platform-specific information that is required by the generic EfiCompatibility code. The functions are invoked via thunking by using **EFI_LEGACY_BIOS_PROTOCOL.FarCall86()** with the 32-bit physical entry point **EFI_COMPATIBILITY16_TABLE**.

The table below lists the Compatibility16 functions that are available.

Table 8 Compatibility16 Functions

Functions	Description
Compatibility16InitializeYourself()	The first function that is invoked and allows the Compatibility16 code to do any initialization. Because EFI performs the equivalent of POST, this invocation is the first time the Compatibility16 code gets control. The region from 0xE0000 to 0xFFFFF is read/write.
Compatibility16UpdateBbs()	Allows the Compatibility16 code to update the CSM’s BBS data structures for any OpROM that hooked INT13, INT18, or INT19. The region from 0xE0000 to 0xFFFFF is read-write.
Compatibility16PrepareToBoot()	Allows the Compatibility16 code to do any last minute cleanup or bookkeeping prior to booting a traditional OS. The region from 0xE0000 to 0xFFFFF is read-write.
Compatibility16Boot()	The last function invoked prior to booting a traditional OS. The region from 0xE0000 to 0xFFFFF is write protected.
Compatibility16RetrieveLastBootDevice()	Retrieves the last boot device priority number. This number allows the CSM to determine the boot device when multiple boot devices exist. The region from 0xE0000 to 0xFFFFF is write protected.
Compatibility16DispatchOprom()	Passes control to the OpROM initialization address under Compatibility16 control. This address allows the Compatibility16 code to re-hook INT13, INT18, and/or INT19 for non-BBS-compliant OpROMs. The region from 0xE0000 to 0xFFFFF is write protected. Note: If the platform allows OpROMs to be placed in the 0xExxxx region, then that region is read/write.
Compatibility16GetTableAddress()	Asks the Compatibility16 to allocate an area of the indicated size in the 0xE0000–0xFFFFF region. The EfiCompatibility code then copies data into that region. The region from 0xE0000 to 0xFFFFF is read-write.
Compatibility16SetKeyboardLeds()	Allows the Compatibility16 code to update any nonstandard data structures with the keyboard LED state. The region from 0xE0000 to 0xFFFFF is read-write.

Compatibility16InstallPciHandler()	Allows the Compatibility16 code to install an IRQ handler for mass storage devices that do not have an OpROM associated with them. An example is Serial ATA (SATA).

2.4.3.1.3 Compatibility16 Function Data Structures

There are two major structures passed from EfiCompatibility to Compatibility16:

- **EFI_TO_COMPATIBILITY16_INIT_TABLE**
- **EFI_TO_COMPATIBILITY16_BOOT_TABLE**

These tables describe the state of the machine at the time the function is issued. **EFI_TO_COMPATIBILITY16_INIT_TABLE** is passed in during the **Compatibility16InitializeYourself()** function.

EFI_TO_COMPATIBILITY16_BOOT_TABLE is passed in during the **Compatibility16PrepareToBoot()** function.

2.4.3.2 Compatibility16 Support

The Compatibility16 support includes all runtime support and all software interrupts, other than OpROMs and traditional hardware interrupt service routines.

2.4.4 CompatibilitySmm

The CompatibilitySmm code is optional. User requirements or traditional features may force it to become a required piece of code. CompatibilitySmm is expected to be chipset and/or platform specific. The following are possible examples:

- System configuration data for INT15 D042 support
- USB legacy support provided for keyboard and mouse
- Update BBS with USB boot devices information

2.4.5 Thunk and Reverse Thunk Overview

Thunk is the code that switches from 32-bit protected environment into the 16-bit real-mode environment. *Reverse thunk* is the code that does the opposite. The code ensures that the 8259 PIC is correct for the environment. This piece of code is arcane.

The transition from EfiCompatibility to Compatibility16 code or to OpROM code requires a “thunk.”. This code does the following:

- Handles any APIC and PIC reprogramming and loading of new GDT and IDT tables.
- Performs the requested action.
- Saves the 16-bit code interrupt state.
- Restores the 32-bit interrupt environment and returns to EFI.

EFI can use either of the following functions to accomplish the thunk:

- **EFI_LEGACY_BIOS_PROTOCOL.Int86()**
- **EFI_LEGACY_BIOS_PROTOCOL.FarCall86()**

The 16-bit code returns to the EFI environment by performing an IRET or FAR RET.

The reverse thunk is similar to a thunk but is used on the 16-bit to 32-bit to 16-bit transitions. There are no defined reverse thunks at this time. Its code is added for completeness.

The figure below shows how the thunk and reverse thunk operate between the 16-bit and 32-bit environments.

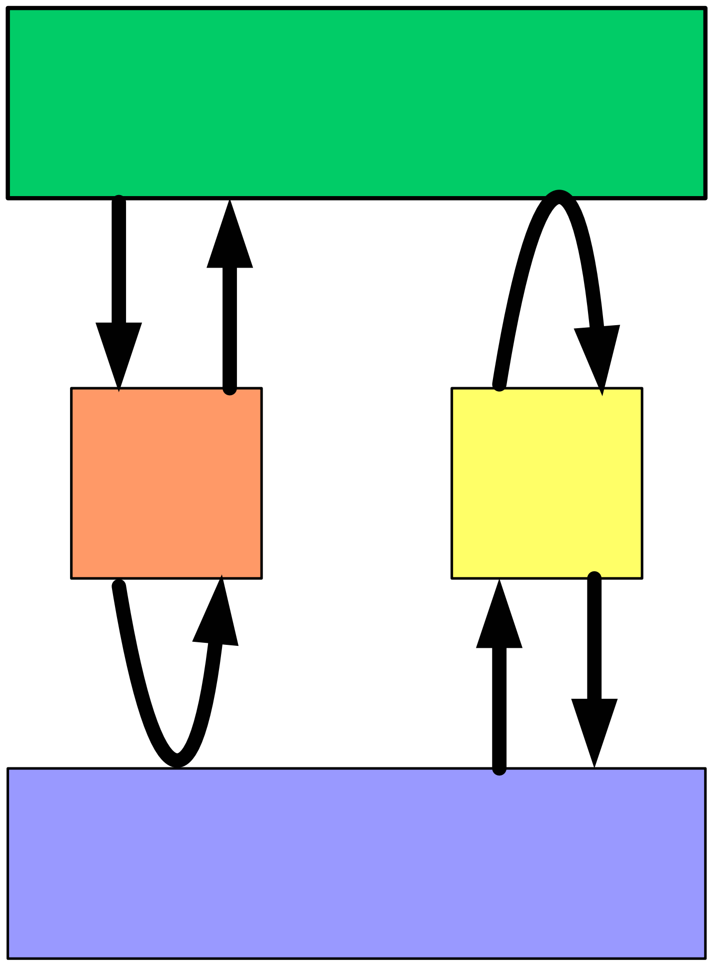

Figure 3 Thunk and Reverse Thunk in a Traditional Code Environment

2.5 Interactions between CSM and Legacy BIOS

2.5.1 BDS and Legacy Drivers

BDS must invoke the CSM by dispatching the Legacy BIOS Protocol if either of the following is true:

- A traditional OpROM is required.
- A traditional boot option is found in the boot sequence

2.5.1.1 Traditional OpROMs

There are two cases where traditional OpROMs are required in an EFI environment, as follows:

No EFI driver exists

There are cases where a required device has no EFI driver but only a traditional OpROM. The normal binding of a device and driver fails and an attempt is made to do the binding via the EfiCompatibility code. This binding is done in the Legacy BIOS Protocol. The thunk driver is bound by the BDS (due to its priority) and the thunk driver calls back into the Legacy BIOS Protocol to load the ROM.

Booting a traditional OS

All devices requiring an OpROM in a traditional BIOS boot will require traditional OpROMs when booting a traditional OS in an EFI environment. This requirement means that there may be an EFI driver and a traditional OpROM for the same device. This transition from EFI to traditional BIOS code is done in **`EFI_LEGACY_BIOS_PROTOCOL.LegacyBoot()`**. This code disconnects all EFI drivers for traditional devices.

2.5.1.2 Determining if Traditional OS Is Present on a Boot Device

The EFI Device Path distinguishes between booting to the following:

- An EFI-aware OS (regardless if the device is EFI or traditional)
- A traditional OS

`EFI_LEGACY_BIOS_PROTOCOL.LegacyBoot()` is used in the latter case. It is recommended that potential removable media boot devices are checked to see if any media are present prior to setting boot devices. This check speeds up the boot time and may prevent a possible system reset. A failed traditional boot will cause a system reset any time control is passed to an OS boot loader and the loader returns back to the BIOS. This reset occurs because EFI might have been corrupted.

2.5.1.3 Determining the Boot Sequence When Traditional OSs Are Involved

The EFI device path determines which of the following is used:

- The normal EFI boot sequence
- The **`EFI_LEGACY_BIOS_PROTOCOL.LegacyBoot()`** sequence

Once it is determined that the **EFI_LEGACY_BIOS_PROTOCOL.LegacyBoot()** is used, then the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.PrepareToBoot()** function is used to order the device boot sequence.

2.5.1.4 Traditional Installation

The traditional BIOS driver is used to abstract the traditional BIOS for EFI. BDS installing the traditional code causes the Legacy BIOS Protocol initialization code to do the following actions:

- Find the Legacy Region Protocol.
- Find the Legacy Interrupt Protocol.
- Find the Legacy BIOS Platform Protocol.
- Find the Legacy 8259 Protocol.
- Allocate the first 4 KB for interrupt vectors and BDA from traditional memory.
- Allocate 0x80000–0x9FFFF for EfiCompatibility usage and for the EBDA.
- Allocate memory for thunk and reverse thunk code.
- Initialize thunk code.
- Initialize the traditional memory map.
- Allocate PMM memory between 1 MB and 16 MB.
- Initialize the BDA and EBDA.
- Locate the firmware volume from which this code was loaded and searches that firmware volume for the Compatibility16 code.
- Determine the size of the Compatibility16 code and from the size calculates the starting address of the Compatibility16 code.
- Make the final destination of the Compatibility16 code read-write and then shadows the Compatibility16 code to the final destination.
- Search for and validate the **EFI_COMPATIBILITY16_TABLE**, saves the Compatibility16 entry point, and updates internal data structures.
- Using the Compatibility16 table function entry point, thunk into the Compatibility16 bit code and request it to perform the function **Compatibility16InitializeYourself()**. The traditional memory map is passed in as a parameter.
- **EFI_COMPATIBILITY16_TABLE** is read to get the Plug and Play installation check address. The internal data structures are updated.
- The Compatibility16 code is then set to read only.
- Install the Legacy BIOS Protocol.
- The Legacy BIOS Protocol returns back to BDS.

2.5.2 16-Bit Traditional Code

The 16-bit traditional code consists of traditional OpROMs and Compatibility16 code. The Compatibility16 code roughly consists of traditional BIOS minus POST and BIOS Setup. It is assembled separately from the EFI code and is linked as a binary module just like an EFI OpROM would be.

See Compatibility16 in CSM Architecture for details on the new Compatibility16 code that is used to interface between the EfiCompatibility portion and the Compatibility16 portion of the traditional BIOS.

2.5.2.1 Legacy BIOS Interface and Functions

There is a table located within the traditional BIOS in either the 0xF000:xxxx or 0xE000:xxxx physical address range. The table is located on a 16-byte boundary and provides the physical address of the entry point for the Compatibility16 Functions.

The Compatibility16 functions provide the platform-specific information that is required by the generic EfiCompatibility code. The functions are invoked via thunking by using **`EFI_LEGACY_BIOS_PROTOCOL.FarCall86()`** with the 32-bit physical entry point **`EFI_COMPATIBILITY16_TABLE`**.

See Compatibility16 Code in Code Definitions for definitions of the Compatibility16 functions and the 32-bit physical entry point.

2.5.2.2 EfiCompatibility to 16-Bit Legacy Transitions

There the following two cases of transitions:

- Thunk
- Reverse thunk

A *thunk* is the transition from EfiCompatibility to 16-bit traditional code and back. A *reverse thunk* is the transition from 16-bit traditional code to EfiCompatibility and back.

See Thunk in CSM Architecture for additional details.

2.5.2.3 EfiCompatibility Drivers

Three drivers are needed to emulate various traditional software INTs, as follows:

- UGA emulation of INT10
- Keyboard emulation of INT16
- Block I/O emulation

The following sections provide more information on these INTs.

2.5.2.4 UGA Emulation of INT10

This situation occurs when traditional OpROMs are to be invoked. The UGA controller is to be placed in VGA emulation mode and a VGA OpROM invoked. This driver must translate EFI console-out data and requests into their VGA equivalent. The following assumptions are used in this document:

- All INT10 functions, both character and dot must be supported.
- The OpROMs may access direct both VGA registers and video memory buffers.
- UGA hardware supports a VGA mode and can be switched between UGA/VGA modes multiple times.

2.5.2.5 Keyboard Emulation of INT16

The Compatibility16 BIOS does not take over USB emulation until the final states of traditional boot. Until that time, INT16 requests must be converted into EFI requests, data received and converted back into INT16 format.

2.5.2.6 Block I/O Emulation

This driver is used when EFI needs to access a traditional floppy or hard disk. It translates EFI block I/O requests into the equivalent INT13 requests.

2.6 Assumptions

2.6.1 External Assumptions

The CSM code makes the following external assumptions:

- When unloaded, EFI device drivers that have EFI OpROMs leave the hardware in a neutral state that allows an equivalent traditional OpROM to be invoked without any adverse device interaction.
- Traditional OpROMs cannot be unloaded and thus leave the hardware in a non-neutral state.
- The UGA hardware is bi-modal, which also supports a VGA emulation mode.
- The UGA OpROM also carries a traditional VGA OpROM.
- Only traditional ACPI-aware OSs are supported.
- Traditional device programming is done either by EFI, EfiCompatibility, or ACPI.
- MS-DOS* boots but there is no guarantee that all DOS programs will work.
- The Legacy BIOS Platform Protocol code, Compatibility16 code, and CompatibilitySmm code (if it exists) are all provided by the same IBV. Using a single IBV ensures consistency and coherency.

2.6.2 Internal Assumptions

The CSM code makes the following internal assumptions:

- The Compatibility16 bit code consists of a traditional runtime BIOS, INT18, and INT19. Compatibility16 does not include 16-bit OpROMs.
- The POST code is removed. The EFI code functions as the traditional BIOS POST equivalent. This assumption presents the minimal space footprint.

2.6.2.1 Compatability16 bit code

The following assumptions pertain to the Compatibility16 bit code except where indicated.

- Runtime text messages are kept to a minimum and are simple ASCII or numerical data. It is considered too expensive, space wise, to carry a display engine for a couple of messages. There is also the coherency of localization between EFI and Compatibility16.
- There is no need for cache control. It is assumed that cache is always enabled or controlled by the OS.
- There are no flash or NVRAM updates, or all updates are done via CompatibilitySmm that is cognizant of EFI firmware volumes and EFI update protocols. There are several reasons for this assumption. Compatibility16 code knows nothing of EFI firmware volumes. Having multiple independent entities trying to maintain flash or NVRAM is guaranteed to introduce system instability.
- There are security problems in having multiple entities maintaining flash or NVRAM.

- Compatibility16 code is chipset hardware neutral.
- CompatibilitySmm code is not chipset hardware neutral.

Having no updates has several ramifications to the Compatibility16 code, as follows:

- No ESCD
- No processor patches
- No update of SMBIOS structures
- No CMOS save to flash

2.6.2.2 SMBIOS

All SMBIOS functions are read only, and both OEMs and manufacturing must use EFI utilities to write asset tags.

SMBIOS version 2.3 is supported in a limited manner, as follows:

- Table entry only.
- No Plug and Play interface.
- Static information only, no flash updates.

2.6.2.3 Other Internal Assumptions

- The boot HDD needs to be INT13 drive 0x80. Other drives can be assigned numbers in any order.
- USB legacy is supported from INT19 on. Pre-INT19 is EFI via any required drivers or via CompatibilitySmm and is CSM implementation specific. This requirement includes keyboard and mouse.
- EFI is sufficient for S3. No legacy code is required.
- EFI drivers, EfiCompatibility drivers, or ACPI ASL are used to program traditional devices. There are no Plug and Play device nodes.
- EFI provides the ASL code.
- There is no APM support. Only ACPI-aware OSs are supported.
- There is no BIOS Setup. EFI provides this functionality.
- There is no POST. EFI provides this functionality.

2.6.3 Design Assumptions

The CSM design assumes the following:

- The major assumption is that large sections of the BIOS IBV's current runtime assembly code will be ported directly to the Compatibility16. EFI interfaces extract information from the EFI modules and translate it into Compatibility16 equivalents.
- The traditional BIOS (Compatibility16) consists of a single module.
- The traditional BIOS is a compiled feature. It is not dynamically controlled via Setup or any other mechanism.
- Compatibility16 does not need to configure motherboard devices. The EFI PEI and DXE phases configure the devices during POST and the ACPI ASL configures devices at runtime. Plug and Play device nodes are not supported. This

assumption implies that the ACPI device configuration selections include entries with and without IRQs.

- Because the Compatibility16 contains traditional devices that use interrupts, it requires an interrupt vector table and interrupts located at the traditional locations. The EFI Interrupt Descriptor Table (IDT) resides at a different address.
- EFI selects the boot ordering, not EfiCompatibility or Compatibility16. EfiCompatibility can change the boot priority.
- Compatibility16 owns resources below 1 MB in memory. An area needs to be reserved for EFI double buffering usage and 0:7C00 needs to be reserved for the boot loader.
- The CSM is invoked in BDS and dispatches the Compatibility16 code.
- If a traditional OS is booted, then all OpROMs must be traditional.
- If an EFI OS is booted, then OpROMs can be either EFI (preferred) or traditional.

2.7 Valid EFI and Legacy Combinations

The table below lists the valid EFI and legacy code combinations:

Table 9 Valid EFI and Legacy Combinations

Video OpROM	Other OpROM	Boot Device OpROM	OS	Valid?
EFI	EFI	EFI	EFI	Compatibility mode not required
EFI	EFI	EFI	Traditional	No
EFI Note1	EFI	Traditional	EFI	Yes
EFI	EFI	Traditional	Traditional	No
EFI Note 1	Traditional	EFI	EFI	Yes
EFI	Traditional	EFI	Traditional	No
EFI Note1	Traditional	Traditional	EFI	Yes
EFI Note1	Traditional	Traditional	Traditional	Yes
Traditional	EFI	EFI	EFI	Yes
Traditional	EFI	EFI	Traditional	No
Traditional	EFI	Traditional	EFI	Yes
Traditional	EFI	Traditional	Traditional	No
Traditional	Traditional	EFI	EFI	Yes
Traditional	Traditional	EFI	Traditional	No
Traditional	Traditional	Traditional	EFI	Yes
Traditional	Traditional	Traditional	Traditional	Yes

Note 1: The EFI UGA video driver must be unloaded and re-invoked in VGA mode with a VGA OpROM.

3
Code Definitions

3.1 Introduction

This section contains definitions of the following protocols, functions, or data types.

Table 10 EFICompatability Code and Compatabiloity16 code

EfiCompatibility Code:	
EFI_LEGACY_BIOS_PROTOCOL	Used to abstract the traditional BIOS for EFI.
EFI_LEGACY_BIOS_PLATFORM_PROTOCOL	Used to abstract the platform-specific traditional hardware and or policy decisions from the generic EfiCompatibility code.
EFI_LEGACY_REGION_PROTOCOL	Used to abstract the hardware control of the OpROM and Compatibility16 region shadowing.
EFI_LEGACY_8259_PROTOCOL	Used to abstract the 8259 PIC.
EFI_LEGACY_INTERRUPT_PROTOCOL	Used to abstract the PIRQ programming from the generic code.
EFI_COMPATIBILITY16_TABLE	A new table introduced to the traditional legacy runtime BIOS for CSM support. Provides the physical address of the entry point for the Compatibility16 functions.
EFI_COMPATIBILITY_FUNCTIONS and the Compatibility16 functions	Allows the EfiCompatibility code to communicate with the Compatibility16 code and are an addition to the traditional BIOS runtime code.

3.2 EfiCompatibility Code

3.2.1 Legacy BIOS Protocol

EFI_LEGACY_BIOS_PROTOCOL

Summary

Abstracts the traditional BIOS from the rest of EFI. The **LegacyBoot()** member function allows the BDS to support booting a traditional OS. EFI thunks drivers that make EFI bindings for BIOS INT services use all the other member functions.

GUID

```
// { DB9A1E3D-45CB-4ABB-853B-E5387FDB2E2D}

#define EFI_LEGACY_BIOS_PROTOCOL_GUID             \

  { 0xdb9a1e3d, 0x45cb, 0x4abb, 0x85, 0x3b, 0xe5, 0x38, 0x7f,
0xdb, 0x2e,  0x2d }
```

Protocol Interface Structure

```
typedef struct _EFI_LEGACY_BIOS_PROTOCOL {

  EFI_LEGACY_BIOS_INT86                        Int86;

  EFI_LEGACY_BIOS_FARCALL86                    FarCall86;

  EFI_LEGACY_BIOS_CHECK_ROM                    CheckPciRom;

  EFI_LEGACY_BIOS_INSTALL_ROM                  InstallPciRom;

  EFI_LEGACY_BIOS_BOOT                         LegacyBoot;

  EFI_LEGACY_BIOS_UPDATE_KEYBOARD_LED_STATUS
UpdateKeyboardLedStatus;

  EFI_LEGACY_BIOS_GET_BBS_INFO                 GetBbsInfo;

  EFI_LEGACY_BIOS_SHADOW_ALL_LEGACY_OPROMS
   ShadowAllLegacyOproms;

  EFI_LEGACY_BIOS_PREPARE_TO_BOOT_EFI          PrepareToBootEFI;

  EFI_LEGACY_BIOS_GET_LEGACY_REGION            GetLegacyRegion;

  EFI_LEGACY_BIOS_COPY_LEGACY_REGION           CopyLegacyRegion;

  EFI_LEGACY_BIOS_BOOT_UNCONVENTIONAL_DEVICE
BootUnconventionalDevice;

} EFI_LEGACY_BIOS_PROTOCOL;
```

Parameters

Int86
: Performs traditional software INT. See the **Int86()** function description.

FarCall86
: Performs a far call into Compatibility16 or traditional OpROM code. See the **FarCall86()** function description.

CheckPciRom
: Checks if a traditional OpROM exists for this device. See the **CheckPciRom()** function description.

InstallPciRom
: Loads a traditional OpROM in traditional OpROM address space. See the **InstallPciRom()** function description.

LegacyBoot
: Boots a traditional OS. See the **LegacyBoot()** function description. See the **PrepareToBootEfi()** function for booting an EFI-aware OS.

UpdateKeyboardLedStatus
: Updates BDA to reflect the current EFI keyboard LED status. See the **UpdateKeyboardLedStatus()** function description.

GetBbsInfo
: Allows an external agent, such as BIOS Setup, to get the BBS data. See the **GetBbsInfo()** function description.

ShadowAllLegacyOproms
: Causes all legacy OpROMs to be shadowed. See the **ShadowAllLegacyOproms()** function description.

PrepareToBootEfi
: Performs all actions prior to boot. Used when booting an EFI-aware OS rather than a legacy OS. See the **PrepareToBootEfi()** function description. See the **LegacyBoot()** function for booting a legacy OS.

GetLegacyRegion
: Allows EFI to reserve an area in the 0xE0000 or 0xF0000 block. See the **GetLegacyRegion()** function description.

CopyLegacyRegion
: Allows EFI to copy data to the area specified by *GetLegacyRegion*. See the **CopyLegacyRegion()** function description.

BootUnconventionalDevice
: Allows the user to boot off an unconventional device such as a PARTIES partition. See the **BootUnconentionalDevice()** function description.

Description

The **EFI_LEGACY_BIOS_PROTOCOL** is used to abstract the traditional BIOS for EFI.

EFI_LEGACY_BIOS_PROTOCOL.Int86()

Summary

Issues a traditional software INT.

Prototype

```
typedef
BOOLEAN
(EFIAPI *EFI_LEGACY_BIOS_INT86) (
  IN   EFI_LEGACY_BIOS_PROTOCOL_    *This,
  IN   UINT8                         BiosInt,
  IN OUT  EFI_IA32_REGISTER_SET     *Regs
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

BiosInt
: The software INT requested.

Regs
: The IA-32 registers. Type **EFI_IA32_REGISTER_SET** is defined in "Related Definitions" below.

Description

This function issues a software INT and gets the results.

Related Definitions

```
//************************************************************
// EFI_IA32_REGISTER_SET
//************************************************************


typedef union {
  EFI_DWORD_REGS  E;
  EFI_WORD_REGS   X;
  EFI_BYTE_REGS   H;
} EFI_IA32_REGISTER_SET;
```

E

Dword registers. Type **`EFI_DWORD_REGS`** is defined below.

X

Word registers. Type **`EFI_WORD_REGS`** is defined below.

H

Byte registers. Type **`EFI_BYTE_REGS`** is defined below.

```
//*******************************************************
// EFI_DWORD_REGS
//*******************************************************

typedef struct {
    UINT32          EAX;
    UINT32          EBX;
    UINT32          ECX;
    UINT32          EDX;
    UINT32          ESI;
    UINT32          EDI;
    EFI_EFLAGS_REG  EFlags;
    UINT16          ES;
    UINT16          CS;
    UINT16          SS;
    UINT16          DS;
    UINT16          FS;
    UINT16          GS;
    UINT32          EBP;
    UINT32          ESP;
} EFI_DWORD_REGS;



//*******************************************************
// EFI_EFLAGS_REG
```

```
//*********************************************************

typedef struct {
    UINT32 CF:1;
    UINT32 Reserved1:1;
    UINT32 PF:1;
    UINT32 Reserved2:1;
    UINT32 AF:1;
    UINT32 Reserved3:1;
    UINT32 ZF:1;
    UINT32 SF:1;
    UINT32 TF:1;
    UINT32 IF:1;
    UINT32 DF:1;
    UINT32 OF:1;
    UINT32 IOPL:2;
    UINT32 NT:1;
    UINT32 Reserved4:2;
    UINT32 VM:1;
    UINT32 Reserved5:14;
} EFI_EFLAGS_REG;


//*********************************************************
// EFI_WORD_REGS
//*********************************************************

typedef struct {
    UINT16          AX;
    UINT16          ReservedAX;
    UINT16          BX;
    UINT16          ReservedBX;
```

```
    UINT16          CX;
    UINT16          ReservedCX;
    UINT16          DX;
    UINT16          ReservedDX;
    UINT16          SI;
    UINT16          ReservedSI;
    UINT16          DI;
    UINT16          ReservedDI;
    EFI_FLAGS_REG  Flags;
    UINT16          ReservedFlags;
    UINT16          ES;
    UINT16          CS;
    UINT16          SS;
    UINT16          DS;
    UINT16          FS;
    UINT16          GS;
    UINT16          BP;
    UINT16          ReservedBP;

    UINT16          SP;
    UINT16          ReservedSP;
} EFI_WORD_REGS;



//*********************************************
// EFI_FLAGS_REG
//*********************************************
typedef struct {
  UINT16      CF:1;
  UINT16      Reserved1:1;
  UINT16      PF:1;
```

```
  UINT16      Reserved2:1;
  UINT16      AF:1;
  UINT16      Reserved3:1;
  UINT16      ZF:1;
  UINT16      SF:1;
  UINT16      TF:1;
  UINT16      IF:1;
  UINT16      DF:1;
  UINT16      OF:1;
  UINT16      IOPL:2;
  UINT16      NT:1;
  UINT16      Reserved4:1;
} EFI_FLAGS_REG;


//*******************************************************
// EFI_BYTE_REGS
//*******************************************************

typedef struct {
    UINT8       AL, AH;
    UINT16      ReservedAX;
    UINT8       BL, BH;
    UINT16      ReservedBX;
    UINT8       CL, CH;
    UINT16      ReservedCX;
    UINT8       DL, DH;
    UINT16      ReservedDX;
} EFI_BYTE_REGS;

#define CARRY_FLAG  0x01
```

Status Codes Returned

FALSE	INT completed. See *Regs* for status.
TRUE	INT was not completed.

EFI_LEGACY_BIOS_PROTOCOL.FarCall86()

Summary

Performs a far call into Compatibility16 or traditional OpROM code.

Prototype

```
typedef
BOOLEAN
(EFIAPI *EFI_LEGACY_BIOS_FARCALL86) (
  IN  EFI_LEGACY_BIOS_PROTOCOL      *This,
  IN  UINT16                        Segment,
  IN  UINT16                        Offset,
  IN  EFI_IA32_REGISTER_SET         *Regs,
  IN  VOID                          *Stack,
  IN  UINTN                         StackSize
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

Segment
: Segment of 16-bit mode call.

Offset
: Offset of 16-bit mode call.

Regs
: The IA-32 registers. Type **EFI_IA32_REGISTER_SET** is defined in **EFI_LEGACY_BIOS_PROTOCOL.Int86()**.

Stack
: Caller-allocated stack that is used to pass arguments.

StackSize
: Size of *Stack* in bytes.

Description

This function performs a far call into Compatibility16 or traditional OpROM code at the specified *Segment:Offset*.

Status Codes Returned

FALSE	**FarCall()** completed. See *Regs* for status.
TRUE	**FarCall()** was not completed.

EFI_LEGACY_BIOS_PROTOCOL.CheckPciRom()

Summary

Tests to see if a traditional PCI ROM exists for this device..

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_CHECK_ROM) (
  IN  EFI_LEGACY_BIOS_PROTOCOL_     *This,
  IN  EFI_HANDLE                    PciHandle
  OUT VOID                          **RomImage, OPTIONAL
  OUT UINTN                         *RomSize, OPTIONAL
  OUT UINTN                         *Flags
  )
```

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

PciHandle

The handle for this device. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

RomImage

Pointer to the ROM image.

RomSize

The size of the ROM image.

Flags

The type of ROM discovered. Multiple bits can be set, as follows:
00 = No ROM
01 = ROM Found
02 = ROM is a valid legacy ROM

Description

This function tests to see if a traditional PCI ROM exists for this device..

Status Codes Returned

EFI_SUCCESS	A traditional OpROM is available for this device.
EFI_UNSUPPORTED	A traditional OpROM is not supported.

EFI_LEGACY_BIOS_PROTOCOL.InstallPciRom()

Summary

Shadows an OpROM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_INSTALL_ROM) (
  IN  EFI_LEGACY_BIOS_PROTOCOL        *This,
  IN  EFI_HANDLE                      PciHandle,
  IN  VOID                            **RomImage,
  OUT UINTN                           *Flags
  OUT UINT8                           *DiskStart, OPTIONAL
  OUT UINT8                           *DiskEnd, OPTIONAL
  OUT VOID                            **RomShadowAddress, OPTIONAL
  OUT UINT32                          *ShadowedRomSize OPTIONAL
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

PciHandle
: The PCI PC-AT* OpROM from this device's ROM BAR will be loaded. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

RomImage
: A PCI PC-AT ROM image. This argument is non-**NULL** if there is no hardware associated with the ROM and thus no *PciHandle*; otherwise it must be **NULL**. An example is the PXE base code.

Flags
: The type of ROM discovered. Multiple bits can be set, as follows:
  00 = No ROM.
  01 = ROM found.
  02 = ROM is a valid legacy ROM.

DiskStart
: Disk number of the first device hooked by the ROM. If *DiskStart* is the same as *DiskEnd*, no disks were hooked.

DiskEnd

Disk number of the last device hooked by the ROM.

RomShadowAddress

Shadow address of PC-AT ROM.

ShadowedRomSize

Size in bytes of *RomShadowAddress*.

Description

This function loads a traditional PC-AT OpROM on the *PciHandle* device and returns information about how many disks were added by the OpROM and the shadow address and size. *DiskStart* and *DiskEnd* are INT13h drive letters. Thus 0x80 is C:.

Status Codes Returned

EFI_SUCCESS	The OpROM was shadowed
EFI_UNSUPPORTED	The *PciHandle* was not found

EFI_LEGACY_BIOS_PROTOCOL.LegacyBoot()

Summary

Boots a traditional OS.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_BOOT) (
  IN  EFI_LEGACY_BIOS_PROTOCOL_  *This,
  IN  BBS_BBS_DEVICE_PATH        *BootOption,
  IN  UINT32                     LoadOptionsSize,
  IN  VOID                       *LoadOptions
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

BootOption
: The EFI device path from BootXXXX variable. Type **BBS_BBS_DEVICE_PATH** is defined in "Related Definitions" below.

LoadOptionSize
: Size of *LoadOption*.

LoadOption
: The load option from BootXXXX variable.

Description

This function attempts to traditionally boot the specified *BootOption*. If the EFI context has been compromised, this function will not return. This procedure is not used for loading an EFI-aware OS off a traditional device. The following actions occur:

- Get EFI SMBIOS data structures, convert them to a traditional format, and copy to Compatibility16.
- Get a pointer to ACPI data structures and copy the Compatibility16 RSD PTR to F0000 block.
- Find the traditional SMI handler from a firmware volume and register the traditional SMI handler with the EFI SMI handler.
- Build onboard IDE information and pass this information to the Compatibility16 code.
- Make sure all PCI Interrupt Line registers are programmed to match 8259.

- Reconfigure SIO devices from EFI mode (polled) into traditional mode (interrupt driven).
- Shadow all PCI ROMs.
- Set up BDA and EBDA standard areas before the legacy boot.
- Construct the Compatibility16 boot memory map and pass it to the Compatibility16 code.
- Invoke the Compatibility16 table function **Compatibility16PrepareToBoot()**. This invocation causes a thunk into the Compatibility16 code, which sets all appropriate internal data structures. The boot device list is a parameter.
- Invoke the Compatibility16 Table function **Compatibility16Boot()**. This invocation causes a thunk into the Compatibility16 code, which does an INT19.
- If the **Compatibility16Boot()** function returns, then the boot failed in a graceful manner—i.e., EFI code is still valid. An ungraceful boot failure causes a reset because the state of EFI code is unknown.

Related Definitions

```
//*******************************************************
// BBS_BBS_DEVICE_PATH
//*******************************************************
#define BBS_DEVICE_PATH           0x05
#define BBS_BBS_DP                0x01

typedef struct _BBS_BBS_DEVICE_PATH {
  EFI_DEVICE_PATH_PROTOCOL        Header;
  UINT16                          DeviceType;
  UINT16                          StatusFlag;
  CHAR8                           String[1];
} BBS_BBS_DEVICE_PATH;
```

Header

The device path header. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

DeviceType

Device type as defined by the BBS Specification. Defined device types are listed in "Related Definitions" in **Compatibility16PrepareToBoot().**

StatusFlag

Status flags as defined by the BBS Specification. Type **BBS_STATUS_FLAGS** is defined in **Compatibility16PrepareToBoot()**.

String

ASCIIZ string that describes the boot device to a user. The length of this string *n* can be determined by subtracting 8 from the *Header.Length* entry.

Status Codes Returned

EFI_DEVICE_ERROR	Failed to boot from any boot device and memory is uncorrupted. Note: This function normally never returns. It will either boot the OS or reset the system if memory has been "corrupted" by loading a boot sector and passing control to it.

EFI_LEGACY_BIOS_PROTOCOL.UpdateKeyboardLedStatus()

Summary

Updates the BDA to reflect status of the Scroll Lock, Num Lock, and Caps Lock keys and LEDs.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_UPDATE_KEYBOARD_LED_STATUS) (
  IN  EFI_LEGACY_BIOS_PROTOCOL   *This,
  IN  UINT8                      Leds
  )
```

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

Leds

Current LED status, as follows:
Bit 0 – Scroll Lock 0 = Off
Bit 1 – Num Lock
Bit 2 – Caps Lock

Description

This function takes the *Leds* input parameter and sets/resets the BDA accordingly. *Leds* is also passed to Compatibility16 code, in case any special processing is required. This function is normally called from EFI Setup drivers that handle user-selectable keyboard options such as boot with NUM LOCK on/off. This function does not touch the keyboard or keyboard LEDs but only the BDA.

Status Codes Returned

EFI_SUCCESS	The BDA was updated successfully.

EFI_LEGACY_BIOS_PROTOCOL.GetBbsInfo()

Summary

Presents BBS information to external agents.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_GET_BBS_INFO) (
  IN    EFI_LEGACY_BIOS_PROTOCOL      *This,
  OUT   UINT16                        *HddCount,
  OUT   HDD_INFO                      **HddInfo,
  OUT   UINT16                        *BbsCount,
  IN OUT  BBS_TABLE                   **BbsTable
  )
```

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

HddCount

Number of **HDD_INFO** structures. Type **HDD_INFO** is defined in "Related Definitions" in **Compatibility16PrepareToBoot()**.

HddInfo

Onboard IDE controller information.

BbsCount

Number of **BBS_TABLE** structures.

BbsTable

BBS entry. Type **BBS_TABLE** is defined in "Related Definitions" in **Compatibility16PrepareToBoot()**.

Description

This function presents the internal BBS data structures to external agents such as BIOS Setup and allows them to assign boot priorities.

Status Codes Returned

EFI_SUCCESS	Tables returned successfully.

EFI_LEGACY_BIOS_PROTOCOL.ShadowAllLegacyOproms()

Summary

Allows external agents to force loading of all legacy OpROMs. This function can be invoked before **GetBbsInfo()** to ensure all devices are counted.

Prototype

```
typedef

EFI_STATUS

(EFIAPI *EFI_LEGACY_BIOS_SHADOW_ALL_LEGACY_OPROMS) (

  IN EFI_LEGACY_BIOS_PROTOCOL *This

  )
```

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

Description

This function forces loading and invocation of the legacy OpROMs, which causes the BBS table to be updated.

Status Codes Returned

EFI_SUCCESS	Tables returned successfully.

EFI_LEGACY_BIOS_PROTOCOL.PrepareToBootEfi()

Summary

This function is called when booting an EFI-aware OS with legacy hard disks. The legacy hard disks may or may not be the boot device but will be accessed by the EFI-aware OS.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_PREPARE_TO_BOOT) (
  IN    EFI_LEGACY_BIOS_PROTOCOL       *This
  OUT   UINT16                         *BbsCount,
  OUT   BBS_TABLE                     **BbsTable

  )
```

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

BbsCount

Number of **BBS_TABLE** structures.

BbsTable

BBS entry. Type **BBS_TABLE** is defined in "Related Definitions" in **Compatibility16PrepareToBoot()**.

Description

This function is called when booting an EFI-aware OS with legacy hard disks. The Compatibility16 code needs to assign drive numbers for BBS entries. The AssignedDriveNumber field in the BBS Table reports back the drive number assigned by the 16-bit CSM. Use **EFI_LEGACY_BIOS_PROTOCOL.LegacyBoot()** for booting a legacy OS.

Status Codes Returned

EFI_SUCCESS	Tables returned successfully.

EFI_LEGACY_BIOS_PROTOCOL.GetLegacyRegion()

Summary

This function is called when EFI needs to reserve an area in the 0xE0000 or 0xF0000 64 KB blocks.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_GET_LEGACY_REGION) (
  IN EFI_LEGACY_BIOS_PROTOCOL          *This,
  IN UINTN                             LegacyMemorySize,
  IN UINTN                             Region,
  IN UINTN                             Alignment,
  OUT VOID                             **LegacyMemoryAddress
  )
```

Parameters

This
Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

LegacyMemorySize
Requested size in bytes of the region.

Region
Requested region.

00 = Either 0xE0000 or 0xF0000 blocks.

Bit0 = 1 Specify 0xF0000 block

Bit1 = 1 Specify 0xE0000 block

Alignment
Bit-mapped value specifying the address alignment of the requested region. The first nonzero value from the right is alignment.

LegacyMemoryAddress
Address assigned.

Description

This function is called when EFI needs to reserve an area in the 0xE0000 or 0xF0000 64 KB blocks. This function may be invoked only once. Use **EFI_LEGACY_BIOS_PROTOCOL.CopyLegacyRegion()** to move data to the returned region.

Status Codes Returned

EFI_SUCCESS	The requested region was assigned.
EFI_ACCESS_DENIED	The function was previously invoked.
Other	The requested region was not assigned.

EFI_LEGACY_BIOS_PROTOCOL.CopyLegacyRegion()

Summary

This function is called when copying data to the region assigned by **EFI_LEGACY_BIOS_PROTOCOL.GetLegacyRegion()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_COPY_LEGACY_REGION) (
  IN EFI_LEGACY_BIOS_PROTOCOL    *This,
  IN UINTN                       LegacyMemorySize,
  IN VOID                        *LegacyMemoryAddress,
  IN VOID                        *LegacyMemorySourceAddress
  )
```

Parameters

This
Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

LegacyMemorySize
Size in bytes of the memory to copy.

LegacyMemoryAddress
The location within the region returned by **EFI_LEGACY_BIOS_PROTOCOL.GetLegacyRegion()**.

LegacyMemorySourceAddress
Source of the data to copy.

Description

This function is called when copying data to the region that was assigned by **GetLegacyRegion()**. It may be invoked multiple times. This function performs boundary checking via information passed into the **EFI_LEGACY_BIOS_PROTOCOL.GetLegacyRegion()**. The user is responsible for any internal checking, if this function is invoked multiple times.

Status Codes Returned

EFI_SUCCESS	The data was copied successfully.
EFI_ACCESS_DENIED	Either the starting or ending address is out of bounds.

EFI_LEGACY_BIOS_PROTOCOL.BootUnconventionalDevice()

Summary

This function is called when either booting to an unconventional device such as a PARTIES partition and/or executing hard disk diagnostics.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_BOOT_UNCONVENTIONAL_DEVICE) (
  IN EFI_LEGACY_BIOS_PROTOCOL         *This,
  IN UDC_ATTRIBUTES                   Attributes,
  IN UINTN                            BbsEntry,
  IN VOID                             *BeerData,
  IN VOID                             *ServiceAreaData
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_BIOS_PROTOCOL** instance.

Attributes
: Flags used to interpret the rest of the input parameters. Type **UDC_ATTRIBUTES** is defined in **Compatibility16PrepareToBoot()**.

BbsEntry
: The zero-based index into the *BbsTable* for the parent device. Type **BBS_TABLE** is defined in **Compatibility16PrepareToBoot()**.

BeerData
: Pointer to the 128 bytes of raw Beer data.

ServiceAreaData
: Pointer to the 64 bytes of raw service area data. It is up to the caller to select the appropriate service area and point to it.

Description

This function is called when booting from an unconventional device such as a PARTIES partition and/or executing hard disk diagnostics. All other *BbsTable* entries are set to ignore and, depending upon *Attributes*, one or two entries are created. If executing hard disk diagnostics, a *BbsEntry* is created and given the highest priority. If booting from an unconventional device, a *BbsEntry* is created and given the highest priority after the diagnostic entry. It is the caller's responsibility to lock all other drives with hidden partitions, if they exist. If an unconventional boot fails, the system is reset to preserve device partition security.

Status Codes Returned

EFI_INVALID_PARAMETER	Either the *Attribute* and/or pointers do not match.

3.2.2 Legacy BIOS Platform Protocol

EFI_LEGACY_BIOS_PLATFORM_PROTOCOL

Note:

The architecture assumes that the creator of this protocol is also the creator of the Compatibility16 code. Having a single creator ensures that IBV-specific code is coherent.

Summary

Abstracts the platform portion of the traditional BIOS. The Legacy BIOS Platform Protocol will match the IBV's traditional BIOS code.

GUID

```
// { 783658A3-4172-4421-A299-E009079C0CB4}

#define EFI_LEGACY_BIOS_PLATFORM_PROTOCOL_GUID         \

  { 0x783658a3, 0x4172, 0x4421, 0xa2, 0x99, 0xe0, 0x9, 0x7,
0x9c, 0xc,
 0xb4 }
```

Protocol Interface Structure

```
typedef struct _EFI_LEGACY_BIOS_PLATFORM_PROTOCOL {

  EFI_LEGACY_BIOS_PLATFORM_GET_PLATFORM_INFO   GetPlatformInfo;

  EFI_LEGACY_BIOS_PLATFORM_GET_PLATFORM_HANDLE
GetPlatformHandle;

  EFI_LEGACY_BIOS_PLATFORM_SMM_INIT            SmmInit;

  EFI_LEGACY_BIOS_PLATFORM_HOOKS               PlatformHooks;

  EFI_LEGACY_BIOS_PLATFORM_GET_ROUTING_TABLE   GetRoutingTable;

  EFI_LEGACY_BIOS_PLATFORM_TRANSLATE_PIRQ      TranslatePirq;

  EFI_LEGACY_BIOS_PLATFORM_PREPARE_TO_BOOT     PrepareToBoot;

} EFI_LEGACY_BIOS_PLATFORM_PROTOCOL;
```

Parameters

GetPlatformInfo

Gets binary data or other platform information. See the **GetPlatformInfo()** function description. There are several subfunctions.

GetPlatformHandle

Returns a buffer of all handles matching the requested subfunction. See the **GetPlatformHandle()** function description. There are several subfunctions.

`SmmInit`

Loads and initializes the traditional BIOS SMM handler. See the **`SmmInit()`** function description.

`PlatformHooks`

Allows platform to perform any required actions after a LegacyBios operation..

`GetRoutingTable`

Gets $PIR table. See the **`GetRoutingTable()`** function description.

`TranslatePirq`

Translates the given PIRQ to the final value after traversing any PCI bridges. See the **`TranslatePirq()`** function description.

`PrepareToBoot`

Final platform function before the system attempts to boot to a traditional OS. See the **`PrepareToBoot()`** function description.

Description

The **`EFI_LEGACY_BIOS_PLATFORM_PROTOCOL`** is used to abstract the platform-specific traditional hardware and or policy decisions from the generic EfiCompatibility code.

EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.GetPlatformInfo()

Summary

Finds the binary data or other platform information. Refer to the sub-functions for additional information.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_PLATFORM_GET_PLATFORM_INFO) (
  IN  EFI_LEGACY_BIOS_PLATFORM_PROTOCOL    *This,
  IN  EFI_GET_PLATFORM_INFO_MODE           Mode,
  IN OUT VOID                              **Table,
  IN OUT UINTN                             *TableSize,
  IN OUT UINTN                             *Location,
  OUT UINTN                                *Alignment,
  IN UINT16                                LegacySegment,
  IN UINT16                                LegacyOffset
  );
```

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

Specifies what data to return.

GetMpTable

GetOemIntData

GetOem16Data

GetOem32Data

GetTpmBinary

GetSystemRom

GetPciExpressBase

GetPlatformPmmSize

GetPlatformEndOpromShadowAddr

Table

Pointer to OEM legacy 16-bit code or data.

TableSize

Size of data.

Location

Location to place table. 0x00 – Either 0xE0000 or 0xF0000 64 KB blocks.

Bit 0 = 1 0xF0000 64 KB block.

Bit 1 = 1 0xE0000 64 KB block.

Multiple bits can be set.

Alignment

Bit-mapped address alignment granularity. The first nonzero bit from the right is the address granularity.

LegacySegment

Segment where EfiCompatibility code will place the table or data.

LegacyOffset

Offset where EfiCompatibility code will place the table or data.

Related Definitions

```
//*********************************************
// EFI_GET_PLATFORM_INFO_MODE
//*********************************************
typedef enum {
  EfiGetPlatformBinaryMpTable       = 0,
  EfiGetPlatformBinaryOemIntData    = 1,
  EfiGetPlatformBinaryOem16Data     = 2,
  EfiGetPlatformBinaryOem32Data     = 3,
  EfiGetPlatformBinaryTpmBinary     = 4,
  EfiGetPlatformBinarySystemRom     = 5,
  EfiGetPlatformPciExpressBase      = 6,
  EfiGetPlatformPmmSize             = 7,
  EfiGetPlatformEndOpromShadowAddr = 8,

} EFI_GET_PLATFORM_INFO_MODE;
```

Description

Refer to the section for each **EFI_GET_PLATFORM_INFO_MODE** description.

Status Codes Returned

EFI_SUCCESS	The data was returned successfully.
EFI_UNSUPPORTED	Mode is not supported on this platform.
EFI_NOT_FOUND	Binary image not found.

3.2.2.1 Mode Values for GetPlatformInfo()

EfiGetPlatformBinaryMpTable

Summary

Returns the multiprocessor (MP) table information

Parameters

This
: Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode
: EfiGetPlatformBinaryMpTable

Table
: Pointer to the MP table.

TableSize
: Size in bytes of the MP table.

Location
: Location to place table. 0x00 – Either 0xE0000 or 0xF0000 64 KB blocks.

Bit 0 = 1 0xF0000 64 KB block.

Bit 1 = 1 0xE0000 64 KB block.

Multiple bits can be set.

Alignment
: Bit-mapped address alignment granularity. The first nonzero bit from the right is the address granularity.

LegacySegment
: Segment where EfiCompatibility code will place the MP table.

LegacyOffset
: Offset where EfiCompatibility code will place the MP table.

Description

This mode is invoked twice. The first invocation has *LegacySegment* and *LegacyOffset* set to 0. The mode returns the MP table address in EFI memory and its size.

The second invocation has *LegacySegment* and *LegacyOffset* set to the location in the 0xF0000 or 0xE0000 block to which the MP table is to be copied. The second invocation allows any MP table address fix-ups to occur in the EFI memory copy of the MP table. The caller, not **EfiGetPlatformBinaryMpTable**, copies the modified MP table to the allocated region in 0xF0000 or 0xE0000 block after the second invocation..

Status Codes Returned

EFI_SUCCESS	The MP table was returned.
EFI_UNSUPPORTED	The MP table is not supported on this platform.

EfiGetPlatformBinaryOemIntData

Summary

Returns any OEM-specific code and/or data.

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiGetPlatformBinaryOemIntData

Table

Pointer to OEM legacy 16-bit code or data.

TableSize

Size of data.

Location

Location to place table. 0x00 – Either 0xE0000 or 0xF0000 64 KB blocks.

Bit 0 = 1 0xF0000 64 KB block.

Bit 1 = 1 0xE0000 64 KB block.

Multiple bits can be set.

Alignment

Bit-mapped address alignment granularity. The first nonzero bit from the right is the address granularity.

LegacySegment

Segment where EfiCompatibility code will place the table or data.

LegacyOffset

Offset where EfiCompatibility code will place the table or data.

Description

This function returns a block of data. The contents and usage is IBV or OEM defined. OEMs or IBVs normally use this function for nonstandard Compatibility16 runtime soft INTs. It is the responsibility of this routine to coalesce multiple OEM 16-bit functions, if they exist, into one coherent package that is understandable by the Compatibility16 code.

This function is invoked twice. The first invocation has *LegacySegment* and *LegacyOffset* set to 0. The function returns the table address in EFI memory and its size.

The second invocation has *LegacySegment* and *LegacyOffset* set to the location in the 0xF0000 or 0xE0000 block to which the data (table) is to be copied. The second invocation allows any data (table) address fix-ups to occur in the EFI memory copy of the table. The caller, not **GetOemIntData()**, copies the modified data (table) to the allocated region in 0xF0000 or 0xE0000 block after the second invocation.

Status Codes Returned

EFI_SUCCESS	The data was returned successfully.
EFI_UNSUPPORTED	Oem INT is not supported on this platform.

Returns any OEM INT-specific code and/or data.

EfiGetPlatformBinaryOem16Data

Summary

Returns any 16-bit OEM-specific code and/or data.

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiGetPlatformBinaryOem16Data

Table

Pointer to OEM legacy 16-bit code or data.

TableSize

Size of data.

Location

Location to place the table. 0x00 – Either 0xE0000 or 0xF0000 64 KB blocks.

Bit 0 = 1 0xF0000 64 KB block.

Bit 1 = 1 0xE0000 64 KB block.

Multiple bits can be set.

Alignment

Bit-mapped address alignment granularity. The first nonzero bit from the right is the address granularity.

LegacySegment

Segment where EfiCompatibility code will place the table or data.

LegacyOffset

Offset where EfiCompatibility code will place the table or data.

Description

This mode returns a block of data. The contents and usage is IBV defined. OEMs or IBVs normally use this mode for nonstandard Compatibility16 runtime 16-bit routines. It is the responsibility of this routine to coalesce multiple OEM 16-bit functions, if they exist, into one coherent package that is understandable by the Compatibility16 code.

An example usage might be a legacy mobile BIOS that has a pre-existing runtime interface to return the battery status to calling applications.

This mode is invoked twice. The first invocation has *LegacySegment* and *LegacyOffset* set to 0. The mode returns the table address in EFI memory and its size.

The second invocation has *LegacySegment* and *LegacyOffset* set to the location in the 0xF0000 or 0xE0000 block to which the table is to be copied. The second invocation allows any table address fix-ups to occur in the EFI memory copy of the table.

The caller, not **`EfiGetPlatformBinaryOem16Data`**, copies the modified table to the allocated region in 0xF0000 or 0xE0000 block after the second invocation..

Status Codes Returned

EFI_SUCCESS	The data was returned successfully.
EFI_UNSUPPORTED	Oem16 is not supported on this platform.

EfiGetPlatformBinaryOem32Data

Summary

Returns any 32-bit OEM-specific code and/or data.

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiGetPlatformBinaryOem32Data

Table

Pointer to OEM legacy 32-bit code or data.

TableSize

Size of data.

Location

Location to place the table. 0x00 – Either 0xE0000 or 0xF0000 64 KB blocks.

Bit 0 = 1 0xF0000 64 KB block.

Bit 1 = 1 0xE0000 64 KB block.

Multiple bits can be set.

Alignment

Bit-mapped address alignment granularity. The first nonzero bit from the right is the address granularity.

LegacySegment

Segment where EfiCompatibility code will place the table or data.

LegacyOffset

Offset where EfiCompatibility code will place the table or data.

Description

This mode returns a block of data. The contents and usage is IBV defined. OEMs or IBVs normally use this mode for nonstandard Compatibility16 runtime 32-bit routines. It is the responsibility of this routine to coalesce multiple OEM 32-bit functions, if they exist, into one coherent package that is understandable by the Compatibility16 code.

An example usage might be a legacy mobile BIOS that has a pre-existing runtime interface to return the battery status to calling applications.

This mode is invoked twice. The first invocation has *LegacySegment* and *LegacyOffset* set to 0. The mode returns the table address in EFI memory and its size.

The second invocation has `LegacySegment` and `LegacyOffset` set to the location in the 0xF0000 or 0xE0000 block to which the table is to be copied. The second invocation allows any table address fix-ups to occur in the EFI memory copy of the table. The caller, not **`EfiGetPlatformBinaryOem32Data`**, copies the modified table to the allocated region in 0xF0000 or 0xE0000 block after the second invocation..

Note: *There are two generic mechanisms by which this mode can be used.*

Mechanism 1: This mode returns the data and the Legacy BIOS Protocol copies the data into the F0000 or E0000 block in the Compatibility16 code. The **`EFI_COMPATIBILITY16_TABLE`** *entries* `Oem32Segment` *and* `Oem32Offset` *can be viewed as two UINT16 entries.*

Mechanism 2: This mode directly fills in the **`EFI_COMPATIBILITY16_TABLE`** *with a pointer to the INT15 E820 region containing the 32-bit code. It returns* **`EFI_UNSUPPORTED`**. *The* **`EFI_COMPATIBILITY16_TABLE`** *entries,* `Oem32Segment` *and* `Oem32Offset`, *can be viewed as two UINT16 entries or as a single UINT32 entry as determined by the IBV.*

Status Codes Returned

EFI_SUCCESS	The data was returned successfully.
EFI_UNSUPPORTED	Oem32 is not supported on this platform.

EfiGetPlatformBinaryTpmBinary

Summary

Gets the TPM (Trusted Platform Module) binary image associated with the onboard TPM device.

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiGetPlatformBinaryTpmBinary

Table

TPM binary image for the onboard TPM device.

TableSize

Size of *BinaryImage* in bytes

Location

Location to place the table. 0x00 – Either 0xE0000 or 0xF0000 64 KB blocks.

Bit 0 = 1 0xF0000 64 KB block.

Bit 1 = 1 0xE0000 64 KB block.

Multiple bits can be set.

Alignment

Bit-mapped address alignment granularity. The first nonzero bit from the right is the address granularity.

LegacySegment

Segment where EfiCompatibility code will place the table or data.

LegacyOffset

Offset where EfiCompatibility code will place the table or data.

Description

This mode returns a TPM binary image for the onboard TPM device.

Status Codes Returned

EFI_SUCCESS	*BinaryImage* is valid.
EFI_UNSUPPORTED	Mode is not supported on this platform.
EFI_NOT_FOUND	No *BinaryImage* was found.

EfiGetPlatformBinarySystemRom

Summary

Finds the Compatibility16 “ROM".

Parameters

This
: Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode
: EfiGetPlatformBinarySystemRom

Table
: System ROM image for the platform

TableSize
: Size of *Table* in bytes

Location
: Ignored

Alignment
: Ignored

LegacySegment
: Ignored

LegacyOffset
: Ignored

Description

The mode finds the Compatibility16 “ROM” image.

Status Codes Returned

EFI_SUCCESS	ROM image found.
EFI_NOT_FOUND	ROM not found.

EfiGetPlatformPciExpressBase

Summary

Gets the PciExpress base address

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiGetPlatformPciExpressBase

Table

Ignored

TableSize

Ignored

Location

Base address of PciExpress memory mapped configuration address space.

Alignment

Ignored

LegacySegment

Ignored

LegacyOffset

Ignored

Description

This mode returns the Base address of PciExpress memory mapped configuration address space

Status Codes Returned

EFI_SUCCESS	Address is valid.
EFI_UNSUPPORTED	System does not PciExpress.

EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.GetPlatformHandle()

Summary

Returns a buffer of handles for the requested sub-function.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_PLATFORM_GET_PLATFORM_HANDLE) (
  IN  EFI_LEGACY_BIOS_PLATFORM_PROTOCOL   *This,
  IN  EFI_GET_PLATFORM_HANDLE_MODE        Mode,
  IN  UINT16                              Type,
  OUT EFI_HANDLE                          **HandleBuffer,
  OUT UINTN                               *HandleCount,
  OUT VOID OPTIONAL                       **AdditionalData
  )
```

Parameters

This

Indicates the EFI_LEGACY_BIOS_PLATFORM_PROTOCOL instance.

Mode

Specifies what handle to return.

GetVgaHandle

GetIdeHandle

GetIsaBusHandle

GetUsbHandle

Type

Handle Modifier – Mode specific

HandleBuffer

Pointer to buffer containing all Handles matching the specified criteria. Handles are sorted in priority order. Type EFI_HANDLE is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification*.

Note: *It is the callers responsibility to save the HandleBuffer if they want to preserve it for future use as any subsequent invocation of this function will destroy the buffer contents.*

HandleCount

Number of handles in HandleBuffer.

AdditionalData

Pointer to additional data returned – mode specific.

Related Definitions

```
//*********************************************
// EFI_GET_PLATFORM_HANDLE_MODE
//*********************************************
typedef enum {
  EfiGetPlatformVgaHandle             = 0,
  EfiGetPlatformIdeHandle             = 1,
  EfiGetPlatformIsaBusHandle          = 2,
  EfiGetPlatformUsbHandle             = 3
} EFI_GET_PLATFORM_HANDLE_MODE;
```

Description

This function returns handles for the specific sub-function specified by Mode.

Status Codes Returned

EFI_SUCCESS	The handle is valid.
EFI_UNSUPPORTED	Mode is not supported on this platform.
EFI_NOT_FOUND	The handle is not known.

3.2.2.2 Mode Values for GetPlatformHandle()

EfiGetPlatformVgaHandle

Summary

Returns the handle for the VGA device that should be used during a Compatibility16 boot.

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiGetPlatformVgaHandle

Type

0x00

HandleBuffer

Buffer of all VGA handles found.

HandleCount

Number of VGA handles found.

AdditionalData

NULL

Description

This mode returns the Compatibility16 policy for the device that should be the VGA controller used during a Compatibility16 boot.

Status Codes Returned

EFI_SUCCESS	The handle is valid.
EFI_UNSUPPORTED	Mode is not supported on this platform.
EFI_NOT_FOUND	The VGA handle is not known.

EfiGetPlatformIdeHandle

Summary

Returns the handle for the IDE controller that should be used during a Compatibility16 boot.

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiGetPlatformIdeHandle

Type

0x00

HandleBuffer

Buffer of all IDE handles found.

HandleCount

Number of IDE handles found.

AdditionalData

Pointer to HddInfo

Information about all onboard IDE controllers. Type **HDD_INFO** is defined in "Related Definitions" in **Compatibility16PrepareToBoot()**.

Description

This mode returns the Compatibility16 policy for the device that should be the IDE controller used during a Compatibility16 boot.

Status Codes Returned

EFI_SUCCESS	The handle is valid.
EFI_UNSUPPORTED	Mode is not supported on this platform.
EFI_NOT_FOUND	The IDE handle is not known.

EfiGetPlatformIsaBusHandle

Summary

Returns the handle for the ISA bus controller that should be used during a Compatibility16 boot.

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiGetPlatformIsaBusHandle

Type

0x00

HandleBuffer

Buffer of all ISA bus handles found.

HandleCount

Number of ISA bus handles found.

AdditionalData

NULL

Description

This mode returns the Compatibility16 policy for the device that should be the ISA bus controller used during a Compatibility16 boot.

Status Codes Returned

EFI_SUCCESS	The handle is valid.
EFI_UNSUPPORTED	Mode is not supported on this platform.
EFI_NOT_FOUND	ISA bus handle is not known.

EfiGetPlatformUsbHandle

Summary

Returns the handle for the USB device that should be used during a Compatibility16 boot.

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiGetPlatformIsaBusHandle

Type

0x00

HandleBuffer

Buffer of all USB handles found.

HandleCount

Number of USB bus handles found.

AdditionalData

NULL

Description

This mode returns the Compatibility16 policy for the device that should be the USB device used during a Compatibility16 boot.

Status Codes Returned

EFI_SUCCESS	The handle is valid.
EFI_UNSUPPORTED	Mode is not supported on this platform.
EFI_NOT_FOUND	USB bus handle is not known.

EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.SmmInit()

Summary

Loads and registers the Compatibility16 handler with the EFI SMM code.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_PLATFORM_SMM_INIT) (
  IN  EFI_LEGACY_BIOS_PLATFORM_PROTOCOL   *This,
  IN  VOID
*EfiToCompatibility16BootTable
  );
```

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

EfiToCompatibility16BootTable

The boot table passed to the Compatibility16. Allows the **SmmInit()** function to update **EFI_TO_COMPATIBILITY16_BOOT_TABLE.SmmTable.**

Description

This function loads and initializes the traditional BIOS SMM handler.

Status Codes Returned

EFI_SUCCESS	The SMM code loaded.
EFI_DEVICE_ERROR	The SMM code failed to load.

EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.PlatformHooks()

Summary

Allows platform to perform any required action after a LegacyBios operation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_PLATFORM_HOOKS) (
  IN  EFI_LEGACY_BIOS_PLATFORM_PROTOCOL   *This,
  IN  EFI_GET_PLATFORM_HOOK_MODE          Mode,
  IN  UINT16                              Type,
  IN  EFI_HANDLE                          DeviceHandle,
 OPTIONAL
  IN OUT UINTN                            *ShadowAddress,
 OPTIONAL
  IN EFI_COMPATIBILITY16_TABLE            Compatibility16Table,
 OPTIONAL
  OUT VOID                                **AdditionalData
 OPTIONAL
  )
```

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

Specifies what handle to return.

PrepareToScanRom

ShadowServiceRoms

AfterRomInit

Type

Mode specific.

DeviceHandle

List of PCI devices in the system. Type EFI_HANDLE is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification*.

Shadowaddress

First free OpROM area, after other OpROMs have been dispatched..

Compatibility16Table
Pointer to the Compatibility16 Table.

AdditionalData
Pointer to additional data returned – mode specific..

Related Definitions

```
//**********************************************
// EFI_GET_PLATFORM_HOOK_MODE
//**********************************************
typedef enum {
  EfiPlatformHookPrepareToScanRom  = 0,
  EfiPlatformHookShadowServiceRoms = 1,
  EfiPlatformHookAfterRomInit      = 2
} EFI_GET_PLATFORM_HOOK_MODE;
```

Note: *Any OEM defined hooks start with 0x8000*

Description

This function invokes the specific sub-function specified by Mode.

Status Codes Returned

EFI_SUCCESS	The operation performed successfully.
EFI_UNSUPPORTED	Mode is not supported on this platform.
EFI_SUCCESS	Mode specific.

3.2.2.3 Mode Values for PlatformHooks()

EfiPrepareToScanRom

Summary

Allows any preprocessing before scanning OpROMs.

Parameters

This
Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode
EfiPlatformHookPrepareToScanRom

Type
0

DeviceHandle
Handle of device OpROM is associated with. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification*.

ShadowAddress
Address where OpROM is shadowed.

Compatibility16Table
NULL

AdditionalData
NULL

Description

This mode allows any preprocessing before scanning OpROMs.

Status Codes Returned

EFI_SUCCESS	All pre-ROM scan operations completed successfully.
EFI_UNSUPPORTED	Do not install OpROM.

EfiShadowServiceRoms

Summary

Shadows legacy OpROMS that may not have a physical device associated with them. Examples are PXE base code and BIS.

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiPlatformHookShadowServiceRoms

Type

0

DeviceHandle

0

ShadowAddress

First free OpROM area, after other OpROMs have been dispatched..

Compatibility16Table

Pointer to the Compatability16 Table.

AdditionalData

NULL

Description

This mode shadows legacy OpROMS that may not have a physical device associated with them. It returns **EFI_SUCCESS** if the ROM was shadowed.

Status Codes Returned

EFI_SUCCESS	The traditional ROM was loaded for this device.
EFI_UNSUPPORTED	Mode is not supported on this platform.

EfiAfterRomInit

Summary

Allows platform to perform any required operation after an OpROM has completed its initialization..

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

Mode

EfiPlatformHookAfterRomInit

Type

0

DeviceHandle

Handle of device OpROM is associated with. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the EFI 1.10 Specification.

ShadowAddress

Address where OpROM is shadowed.

Compatibility16Table

NULL

AdditionalData

NULL

Description

This mode allows platform to perform any required operation after an OpROM has completed its initialization.

Status Codes Returned

EFI_SUCCESS	The traditional ROM was loaded for this device.
EFI_UNSUPPORTED	Mode is not supported on this platform.

EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.GetRoutingTable()

Summary

Returns information associated with PCI IRQ routing.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_PLATFORM_GET_ROUTING_TABLE) (
  IN  EFI_LEGACY_BIOS_PLATFORM_PROTOCOL   *This,
  OUT VOID                             **RoutingTable,
  OUT UINTN                            *RoutingTableEntries,
  OUT VOID                             **LocalPirqTable, OPTIONAL
  OUT UINTN                            *PirqTableSize, OPTIONAL
  OUT VOID                             **LocalIrqPriorityTable, OPTIONAL
  OUT UINTN                            *IrqPriorityTableEntries OPTIONAL
  )
```

Parameters

This
Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

RoutingTable
Pointer to the PCI IRQ routing table. This location is the $PIR table minus the header. The contents are described by the PCI IRQ Routing Table Specification and consist of *RoutingTableEntries* of **EFI_LEGACY_IRQ_ROUTING_ENTRY**. Type **EFI_LEGACY_IRQ_ROUTING_ENTRY** is defined in “Related Definitions” below.

RoutingTableEntries
Number of entries in the PCI IRQ routing table.

LocalPirqTable
$PIR table. It consists of **EFI_LEGACY_PIRQ_TABLE_HEADER**, immediately followed by *RoutingTable*. Type **EFI_LEGACY_PIRQ_TABLE_HEADER** is defined in "Related Definitions" below.

PirqTableSize
Size of $PIR table.

LocalIrqPriorityTable
A priority table of IRQs to assign to PCI. This table consists of *IrqPriorityTableEntries* of

EFI_LEGACY_IRQ_PRIORITY_TABLE_ENTRY and is used to prioritize the allocation of IRQs to PCI. Type **EFI_LEGACY_IRQ_PRIORITY_TABLE_ENTRY** is defined in "Related Definitions" below.

IrqPriorityTableEntries

Number of entries in the priority table.

Description

This function returns the following information associated with PCI IRQ routing:

- An IRQ routing table and number of entries in the table
- The $PIR table and its size
- A list of PCI IRQs and the priority order to assign them

Related Definitions

```
//**********************************************
// EFI_LEGACY_IRQ_ROUTING_ENTRY
//**********************************************


typedef struct {
  UINT8                     Bus;
  UINT8                     Device;
  EFI_LEGACY_PIRQ_ENTRY     PirqEntry[4];
  UINT8                     Slot;
  UINT8                     Reserved;
} EFI_LEGACY_IRQ_ROUTING_ENTRY;
```

Bus

PCI bus of the entry.

Device

PCI device of this entry.

PirqEntry

An IBV value and IRQ mask for PIRQ pins A through D. Type **EFI_LEGACY_PIRQ_ENTRY** is defined below.

Slot

If nonzero, the slot number assigned by the board manufacturer.

Reserved

Reserved for future use.

```
//*********************************************
// EFI_LEGACY_PIRQ_ENTRY
//*********************************************

typedef struct {
  UINT8                     Pirq;
  UINT16                    IrqMask;
} EFI_LEGACY_PIRQ_ENTRY;
```

Pirq
If nonzero, a value assigned by the IBV.

IrqMask
If nonzero, the IRQs that can be assigned to this device.

```
//***************************************
// EFI_LEGACY_PIRQ_TABLE_HEADER
//***************************************

typedef struct {
  UINT32                    Signature;
  UINT8                     MinorVersion;
  UINT8                     MajorVersion;
  UINT16                    TableSize;
  UINT8                     Bus;
  UINT8                     DevFun;
  UINT16                    PciOnlyIrq;
  UINT16                    CompatibleVid;
  UINT16                    CompatibleDid;
  UINT32                    Miniport;
  UINT8                     Reserved[11];
  UINT8                     Checksum;
} EFI_LEGACY_PIRQ_TABLE_HEADER;
```

Signature

“$PIR”.

MinorVersion

0x00.

MajorVersion

0x01 for table version 1.0.

TableSize

0x20 + *RoutingTableEntries* * 0x10.

Bus

PCI interrupt router bus.

DevFunc

PCI interrupt router device/function.

PciOnlyIrq

If nonzero, bit map of IRQs reserved for PCI.

CompatibleVid

Vendor ID of a compatible PCI interrupt router.

CompatibleDid

Device ID of a compatible PCI interrupt router.

Minport

If nonzero, a value passed directly to the IRQ miniport’s Initialize function.

Reserved

Reserved for future usage.

Checksum

This byte plus the sum of all other bytes in the *LocalPirqTable* equal 0x00.

```
//*********************************************
// EFI_LEGACY_IRQ_PRIORITY_TABLE_ENTRY
//*********************************************

typedef struct {
  UINT8                    Irq;
  UINT8                    Used;
} EFI_LEGACY_IRQ_PRIORITY_TABLE_ENTRY;
```

Irq

IRQ for this entry.

Used

Status of this IRQ.

PCI_UNUSED 0x00 – This IRQ has not been assigned to PCI.

PCI_USED 0xFF – This IRQ has been assigned to PCI.

LEGACY_USED 0xFE – This IRQ has been used by an SIO legacy device and cannot be used by PCI.

Status Codes Returned

EFI_SUCCESS	Data was returned successfully.

EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.TranslatePirq()

Summary

Translates the given PIRQ accounting for bridges.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_PLATFORM_TRANSLATE_PIRQ) (
  IN   EFI_LEGACY_BIOS_PLATFORM_PROTOCOL  *This,
  IN   UINTN                              PciBus,
  IN   UINTN                              PciDevice,
  IN   UINTN                              PciFunction,
  IN   OUT UINT8                          *Pirq,
  OUT UINT8                               *PciIrq
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

PciBus
: PCI bus number for this device.

PciDevice
: PCI device number for this device.

PciFunction
: PCI function number for this device.

Pirq
: The PIRQ. PIRQ A = 0, PIRQ B = 1, and so on.

PirqIrq
: IRQ assigned to the indicated PIRQ.

Description

This function translates the given PIRQ back through all buses, if required, and returns the true PIRQ and associated IRQ.

Status Codes Returned

EFI_SUCCESS	The PIRQ was translated.

EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.PrepareToBoot()

Summary

Attempts to boot a traditional OS.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_BIOS_PLATFORM_PREPARE_TO_BOOT) (
  IN  EFI_LEGACY_BIOS_PLATFORM_PROTOCOL  *This,
  IN  BBS_BBS_DEVICE_PATH                *BbsDevicePath,
  IN  VOID                               *BbsTable,
  IN  UINT32                             LoadOptionsSize,
  IN  VOID                               *LoadOptions,
  IN  VOID                               *EfiToLegacyBootTable
  )
```

Parameters

This

Indicates the **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL** instance.

BbsDevicePath

EFI Device Path from BootXXXX variable. Type **BBS_BBS_DEVICE_PATH** is defined in **EFI_LEGACY_BIOS_PROTOCOL.LegacyBoot()**.

BbsTable

A list of BBS entries of type **BBS_TABLE**. Type **BBS_TABLE** is defined in **Compatibility16PrepareToBoot()**.

LoadOptionsSize

Size of *LoadOption* in bytes.

LoadOptions

LoadOption from BootXXXX variable.

EfiToLegacyBootTable

Pointer to **EFI_TO_COMPATIBILITY16_BOOT_TABLE**. Type **EFI_TO_COMPATIBILITY16_BOOT_TABLE** is defined in **Compatibility16PrepareToBoot()**.

Description

This function assigns priorities to BBS table entries.

Status Codes Returned

EFI_SUCCESS	Ready to boot.

3.2.3 Legacy Region Protocol

EFI_LEGACY_REGION_PROTOCOL

Summary

Abstracts the hardware control of the physical address region 0xC0000–0xFFFFF for the traditional BIOS.

GUID

```
// { 0FC9013A-0568-4BA9-9B7E-C9C390A6609B }

#define EFI_LEGACY_REGION_PROTOCOL_GUID          \

  { 0xfc9013a, 0x568, 0x4ba9, 0x9b, 0x7e, 0xc9, 0xc3, 0x90,
0xa6, 0x60,
0x9b }
```

Protocol Interface Structure

```
typedef struct _EFI_LEGACY_REGION_PROTOCOL {

  EFI_LEGACY_REGION_DECODE                  Decode;

  EFI_LEGACY_REGION_LOCK                    Lock;

  EFI_LEGACY_REGION_BOOT_LOCK               BootLock;

  EFI_LEGACY_REGION_UNLOCK                  UnLock;

} EFI_LEGACY_REGION_PROTOCOL;
```

Parameters

Decode
: Specifies a region for the chipset to decode. See the **Decode()** function description.

Lock
: Makes the specified OpROM region read only or locked. See the **Lock()** function description.

BootLock
: Sets a region to read only and ensures tat flash is locked from inadvertent modification. See the **BootLock()** function description.

Unlock
: Makes the specified OpROM region read-write or unlocked. See the **Unlock()** function description.

Description

The **EFI_LEGACY_REGION_PROTOCOL** is used to abstract the hardware control of the OpROM and Compatibility16 region shadowing.

EFI_LEGACY_REGION_PROTOCOL.Decode()

Summary

Sets hardware to decode or not decode a region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION_DECODE) (
  IN  EFI_LEGACY_REGION_PROTOCOL   *This,
  IN  UINT32                       Start,
  IN  UINT32                       Length,
  IN  BOOLEAN                      *On
  );
```

Parameters

This

Indicates the **EFI_LEGACY_REGION_PROTOCOL** instance

Start

Start of region to decode.

Length

Size in bytes of the region.

On

Decode/nondecode flag.

Description

This function sets the hardware to either decode or not decode a region within 0xC0000 to 0xFFFFF.

Status Codes Returned

EFI_SUCCESS	Decode range successfully changed.

EFI_LEGACY_REGION_PROTOCOL.Lock()

Summary

Sets a region to read only.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION_LOCK) (
  IN EFI_LEGACY_REGION_PROTOCOL   *This,
  IN  UINT32                      Start,
  IN  UINT32                      Length,
  OUT UINT32                      *Granularity  OPTIONAL
  );
```

Parameters

This
: Indicates the **EFI_LEGACY_REGION_PROTOCOL** instance

Start
: Start of the region to lock.

Length
: Length of the region.

Granularity
: Lock attribute affects this granularity in bytes.

Description

This function makes a region from 0xC0000 to 0xFFFFF read only.

Status Codes Returned

EFI_SUCCESS	The region was made read only.

EFI_LEGACY_REGION_PROTOCOL.BootLock()

Summary

Sets a region to read only and ensures that flash is locked from being inadvertently modified.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION_BOOT_LOCK) (
  IN  EFI_LEGACY_REGION_PROTOCOL   *This,
  IN  UINT32                       Start,
  IN  UINT32                       Length,
  OUT UINT32                       *Granularity  OPTIONAL
  );
```

Parameters

This
: Indicates the **EFI_LEGACY_REGION_PROTOCOL** instance

Start
: Start of the region to lock.

Length
: Length of the region

Granularity
: Lock attribute affects this granularity in bytes.

Description

This function makes a region from 0xC0000 to 0xFFFFF read only and prevents writes to any alias regions.

Status Codes Returned

EFI_SUCCESS	The region was made read only and flash is locked.

EFI_LEGACY_REGION_PROTOCOL.UnLock()

Summary

Sets a region to read-write.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION_UNLOCK) (
  IN  EFI_LEGACY_REGION_PROTOCOL   *This,
  IN  UINT32                       Start,
  IN  UINT32                       Length,
  OUT UINT32                       *Granularity  OPTIONAL
  );
```

Parameters

This
Indicates the **EFI_LEGACY_REGION_PROTOCOL** instance

Start
Start of the region to lock.

Length
Length of the region

Granularity
Lock attribute affects this granularity in bytes.

Description

This function makes a region from 0xC0000 to 0xFFFFF read/write.

Note: *This function might have to prevent writes to RAM in the region from propagating to the NVRAM, if other drivers do not. An IA-32 example is where a write to 0xFxxxx can also propagate to 0xFFFFFxxxx and an innocent data pattern can mimic a NVRAM write or erase sequence.*

Status Codes Returned

EFI_SUCCESS	The region was successfully made read-write.

3.2.4 Legacy 8259 Protocol

EFI_LEGACY_8259_PROTOCOL

Summary

Abstracts the 8259 and APIC hardware control between EFI usage and Compatibility16 usage.

GUID

```
// { 38321DBA-4FE0-4E17-8AEC-413055EAEDC1 }
#define EFI_LEGACY_8259_PROTOCOL_GUID            \
  { 0x38321dba, 0x4fe0, 0x4e17,  0x8a, 0xec, 0x41, 0x30, 0x55, \
    0xea, 0xed, 0xc1 }
```

Protocol Interface Structure

```
typedef struct _EFI_LEGACY_8259_PROTOCOL {
  EFI_LEGACY_8259_SET_ VECTOR_BASE        SetVectorBase;
  EFI_LEGACY_8259_GET_MASK                GetMask;
  EFI_LEGACY_8259_SET_MASK                SetMask;
  EFI_LEGACY_8259_SET_MODE                SetMode;
  EFI_LEGACY_8259_GET_VECTOR              GetVector;
  EFI_LEGACY_8259_ENABLE_IRQ              EnableIrq;
  EFI_LEGACY_8259_DISABLE_IRQ             DisableIrq;
  EFI_LEGACY_8259_GET_INTERRUPT_LINE      GetInterruptLine;
  EFI_LEGACY_8259_END_OF_INTERRUPT        EndOfInterrupt
} EFI_LEGACY_8259_PROTOCOL;
```

Parameters

SetVectorBase

Sets the vector bases for master and slave PICs. See the **SetVectorBase()** function description.

GetMask

Gets IRQ and edge/level masks for 16-bit real mode and 32-bit protected mode. See the **GetMask()** function description.

SetMask

Sets the IRQ and edge\level masks for 16-bit real mode and 32-bit protected mode. See the **`Setmask()`** function description.

`SetMode`
Sets PIC mode to 16-bit real mode or 32-bit protected mode. See the **`SetMode()`** function description.

`GetVector`
Gets the base vector assigned to an IRQ. See the **`GetVector()`** function description.

`EnableIrq`
Enables an IRQ. See the **`EnableIrq()`** function description.

`DisableIrq`
Disables an IRQ. See the **`DisableIrq()`** function description.

`GetInterruptLine`
Gets an IRQ that is assigned to a PCI device. See the **`GetInterruptLine()`** function description.

`EndOfInterrupt`
Issues the end of interrupt command. See the **`EndOfInterrupt()`** function description.

Description

The **`EFI_LEGACY_8259_PROTOCOL`** is used to abstract the 8259 PIC.

EFI_LEGACY_8259_PROTOCOL.SetVectorBase()

Summary

Sets the base address for the 8259 master and slave PICs.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_8259_SET_VECTOR_BASE) (
  IN EFI_LEGACY_8259_PROTOCOL   *This,
  IN UINT8                      MasterBase,
  IN UINT8                      SlaveBase
  )
```

Parameters

This
Indicates the **EFI_LEGACY_8259_PROTOCOL** instance.

MasterBase
Interrupt vectors for IRQ0–IRQ7.

SlaveBase
Interrupt vectors for IRQ8–IRQ15.

Description

This function sets the 8259 master and slave address that maps the IRQ to the processor interrupt vector number.

Status Codes Returned

EFI_SUCCESS	The 8259 PIC was programmed successfully.
EFI_DEVICE_ERROR	There was an error while writing to the 8259 PIC.

EFI_LEGACY_8259_PROTOCOL.GetMask()

Summary

Gets the current 16-bit real mode and 32-bit protected-mode IRQ masks.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_8259_GET_MASK) (
  IN  EFI_LEGACY_8259_PROTOCOL_*This,
  OUT  UINT16                    *LegacyMask, OPTIONAL
  OUT  UINT16                    *LegacyEdgeLevel, OPTIONAL
  OUT  UINT16                    *ProtectedMask, OPTIONAL
  OUT  UINT16                    *ProtectedEdgeLevel OPTIONAL
  )
```

Parameters

This
Indicates the **EFI_LEGACY_8259_PROTOCOL** instance.

LegacyMask
16-bit mode interrupt mask for IRQ0–IRQ15.

LegacyEdgeLevel
16-bit mode edge/level mask for IRQ0–IRQ15.

ProtectedMask
32-bit mode interrupt mask for IRQ0–IRQ15.

ProtectedEdgeLevel
32-bit mode edge/level mask for IRQ0–IRQ15.

Description

This function gets the current settings of the interrupt mask and edge/level mask for the 16-bit real-mode operation and 32-bit protected-mode operation.

Status Codes Returned

EFI_SUCCESS	The 8259 PIC was programmed successfully.
EFI_DEVICE_ERROR	There was an error while reading the 8259 PIC.

EFI_LEGACY_8259_PROTOCOL.SetMask()

Summary

Sets the current 16-bit real mode and 32-bit protected-mode IRQ masks.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_8259_SET_MASK) (
  IN EFI_LEGACY_8259_PROTOCOL_*This,
  INT  UINT16                 *LegacyMask, OPTIONAL
  IN  UINT16                  *LegacyEdgeLevel, OPTIONAL
  IN  UINT16                  *ProtectedMask, OPTIONAL
  IN  UINT16                  *ProtectedEdgeLevel OPTIONAL
  )
```

Parameters

This
Indicates the **EFI_LEGACY_8259_PROTOCOL** instance.

LegacyMask
16-bit mode interrupt mask for IRQ0–IRQ15.

LegacyEdgeLevel
16-bit mode edge/level mask for IRQ0–IRQ15.

ProtectedMask
32-bit mode interrupt mask for IRQ0–IRQ15.

ProtectedEdgeLevel
32-bit mode edge/level mask for IRQ0–IRQ15.

Description

This function sets the current settings of the interrupt mask and edge/level mask for the 16-bit real-mode operation and 32-bit protected-mode operation.

Status Codes Returned

EFI_SUCCESS	The 8259 PIC was programmed successfully.
EFI_DEVICE_ERROR	There was an error while reading the 8259 PIC.

EFI_LEGACY_8259_PROTOCOL.SetMode()

Summary

Sets the mode of the PICs.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_8259_SET_MODE) (
  IN  EFI_LEGACY_8259_PROTOCOL    *This,
  IN  EFI_8259_MODE               Mode,
  IN UINT16                       *Mask, OPTIONAL
  IN UINT16                       *EdgeLevel OPTIONAL
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_8259_PROTOCOL** instance.

Mode
: 16-bit real or 32-bit protected mode. Type **EFI_8259_MODE** is defined in "Related Definitions" below.

Mask
: The value with which to set the interrupt mask.

EdgeLevel
: The value with which to set the edge/level mask.

Description

This function switches from one mode to the other mode. This procedure is not to be invoked multiple times for changing masks in the same mode but changing masks. Use the **EFI_LEGACY_8259_PROTOCOL.SetMask()** function instead.

Related Definitions

```
//****************************************************
// EFI_8259_MODE
//****************************************************
typedef enum {
  Efi8259LegacyMode,
```

```
  Efi8259ProtectedMode,
  Efi8259MaxMode
} EFI_8259_MODE;
```

Status Codes Returned

EFI_SUCCESS	The mode was set successfully.
EFI_INVALID_PARAMETER	The mode was not set.

EFI_LEGACY_8259_PROTOCOL.GetVector()

Summary

Translates the IRQ into a vector.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_8259_GET_VECTOR) (
  IN EFI_LEGACY_8259_PROTOCOL   *This,
  IN EFI_8259_IRQ               Irq
  OUT UINT8                     *Vector
)
```

Parameters

This
: Indicates the **EFI_LEGACY_8259_PROTOCOL** instance.

Irq
: IRQ0–IRQ15. Type **EFI_8259_IRQ** is defined in "Related Definitions" below.

Vector
: The vector that is assigned to the IRQ.

Description

This function retrieves the vector that is assigned to the IRQ.

Related Definitions

```
//********************************************
// EFI_8259_IRQ
//********************************************
typedef enum {
  Efi8259Irq0,
  Efi8259Irq1,
  Efi8259Irq2,
  Efi8259Irq3,
  Efi8259Irq4,
  Efi8259Irq5,
```

```
  Efi8259Irq6,
  Efi8259Irq7,
  Efi8259Irq8,
  Efi8259Irq9,
  Efi8259Irq10,
  Efi8259Irq11,
  Efi8259Irq12,
  Efi8259Irq13,
  Efi8259Irq14,
  Efi8259Irq15,
  Efi8259IrqMax
} EFI_8259_IRQ;
```

Status Codes Returned

EFI_SUCCESS	The *Vector* that matches *Irq* was returned.
EFI_INVALID_PARAMETER	*Irq* is not valid.

EFI_LEGACY_8259_PROTOCOL.EnableIrq()

Summary

Enables the specified IRQ.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_8259_ENABLE_IRQ) (
  IN  EFI_LEGACY_8259_PROTOCOL   *This,
  IN  EFI_8259_IRQ              Irq,
  IN  BOOLEAN                   LevelTriggered
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_8259_PROTOCOL** instance.

Irq
: 8259 IRQ0–IRQ15. Type **EFI_8259_IRQ** is defined in **EFI_LEGACY_8259_PROTOCOL.GetVector()**.

LevelTriggered
: 0 = Edge triggered; 1 = Level triggered.

Description

This function enables the specified *Irq* by unmasking the interrupt in the 32-bit mode environment's 8259 PIC.

Status Codes Returned

EFI_SUCCESS	The *Irq* was enabled on the 8259 PIC.
EFI_INVALID_PARAMETER	The *Irq* is not valid.

EFI_LEGACY_8259_PROTOCOL.DisableIrq()

Summary

Disables the specified IRQ.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_8259_DISABLE_IRQ) (
  IN  EFI_LEGACY_8259_PROTOCOL    *This,
  IN  EFI_8259_IRQ                Irq
  )
```

Parameters

This

Indicates the **EFI_LEGACY_8259_PROTOCOL** instance.

Irq

8259 IRQ0–IRQ15. Type **EFI_8259_IRQ** is defined in **EFI_LEGACY_8259_PROTOCOL.GetVector()**.

Description

This function disables the specified *Irq* by masking the interrupt in the 32-bit mode environment's 8259 PIC.

Status Codes Returned

EFI_SUCCESS	The *Irq* was disabled on the 8259 PIC.
EFI_INVALID_PARAMETER	The *Irq* is not valid.

EFI_LEGACY_8259_PROTOCOL.GetInterruptLine()

Summary

Reads the PCI configuration space to get the interrupt number that is assigned to the card.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_8259_GET_INTERRUPT_LINE) (
  IN  EFI_LEGACY_8259_PROTOCOL   *This,
  IN  EFI_HANDLE                 PciHandle,
  OUT UINT8                      *Vector
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_8259_PROTOCOL** instance.

PciHandle
: PCI function for which to return the vector. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Vector
: IRQ number that corresponds to the interrupt line.

Description

This function reads the PCI configuration space to get the interrupt number that is assigned to the card. *PciHandle* represents a PCI configuration space of a PCI function. *Vector* represents the interrupt pin (from the PCI configuration space) and it is the data that is programmed into the interrupt line (from the PCI configuration space) register.

Status Codes Returned

EFI_SUCCESS	The interrupt line value was read successfully.

EFI_LEGACY_8259_PROTOCOL.EndOfInterrupt()

Summary

Issues the End of Interrupt (EOI) commands to PICs.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_8259_END_OF_INTERRUPT) (
  IN  EFI_LEGACY_8259_PROTOCOL   *This,
  IN  EFI_8259_IRQ               Irq
  )
```

Parameters

This

Indicates the **EFI_LEGACY_8259_PROTOCOL** instance.

Irq

The interrupt for which to issue the EOI command. Type **EFI_8259_IRQ** is defined in **EFI_LEGACY_8259_PROTOCOL.GetVector()**.

Description

This function issues the end of interrupt commands to PICs.

Status Codes Returned

EFI_SUCCESS	The EOI command was issued.
EFI_INVALID_PARAMETER	The *Irq* is not valid.

3.2.5 Legacy Interrupt Protocol

EFI_LEGACY_INTERRUPT_PROTOCOL

Summary

Abstracts the PIRQ programming from the generic EFI Compatibility Support Modules (CSMs).

GUID

```
//  { 31CE593D-108A-485D-ADB2-78F21F2966BE }

#define EFI_LEGACY_INTERRUPT_PROTOCOL_GUID       \

  { 0x31ce593d, 0x108a, 0x485d,  0xad, 0xb2, 0x78, 0xf2, 0x1f,
0x29, 0x66, 0xbe }
```

Protocol Interface Structure

```
typedef struct _EFI_LEGACY_INTERRUPT_PROTOCOL {

  EFI_LEGACY_INTERRUPT_GET_NUMBER_PIRQS        GetNumberPirqs;

  EFI_LEGACY_INTERRUPT_GET_LOCATION            GetLocation;

  EFI_LEGACY_INTERRUPT_READ_PIRQ               ReadPirq;

  EFI_LEGACY_INTERRUPT_WRITE_PIRQ              WritePirq;

} EFI_LEGACY_INTERRUPT_PROTOCOL;
```

Parameters

GetNumberPirqs
: Gets the number of PIRQs supported. See the **GetNumberPirqs()** function description.

GetLocation
: Gets the PCI bus, device, and function that associated with this protocol. See the **GetLocation()** function description.

ReadPirq
: Reads the indicated PIRQ register. See the **ReadPirq()** function description.

WritePirq
: Writes to the indicated PIRQ register. See the **WritePirq()** function description.

Description

The **EFI_LEGACY_INTERRUPT_PROTOCOL** is used to abstract the PIRQ programming from the generic code.

EFI_LEGACY_INTERRUPT_PROTOCOL.GetNumberPirqs()

Summary

Gets the number of PIRQs that this hardware supports.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_INTERRUPT_GET_NUMBER_PIRQS) (
  IN  EFI_LEGACY_INTERRUPT_PROTOCOL   *This,
  OUT UINT8                           *NumberPirqs
  )
```

Parameters

This
Indicates the **EFI_LEGACY_INTERRUPT_PROTOCOL** instance.

NumberPirqs
Number of PIRQs that are supported.

Description

This function gets the number of PIRQs that are supported by the hardware.

Status Codes Returned

EFI_SUCCESS	The number of PIRQs was returned successfully.

EFI_LEGACY_INTERRUPT_PROTOCOL.GetLocation()

Summary

Gets the PCI location associated with this protocol.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_INTERRUPT_GET_LOCATION) (
  IN  EFI_LEGACY_INTERRUPT_PROTOCOL  *This,
  OUT UINT8                          *Bus,
  OUT UINT8                          *Device,
  OUT UINT8                          *Function
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_INTERRUPT_PROTOCOL** instance.

Bus
: PCI bus number of this device.

Device
: PCI device number of this device.

Function
: PCI function number of this device.

Description

This function gets the PCI location of the device supporting this protocol.

Status Codes Returned

EFI_SUCCESS	The PCI bus number was returned successfully.

EFI_LEGACY_INTERRUPT_PROTOCOL.ReadPirq()

Summary

Reads the given PIRQ register and returns the IRQ that is assigned to it.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_INTERRUPT_READ_PIRQ) (
  IN   EFI_LEGACY_INTERRUPT_PROTOCOL   *This,
  IN   UINT8                           PirqNumber,
  OUT  UINT8                           *PirqData
  )
```

Parameters

This
: Indicates the **EFI_LEGACY_INTERRUPT_PROTOCOL** instance.

PirqNumber
: PIRQ A = 0, PIRQ B = 1, and so on.

PirqData
: IRQ assigned to this PIRQ

Description

This function reads the indicated PIRQ register and returns the IRQ that is assigned to it.

Status Codes Returned

EFI_SUCCESS	The data was returned successfully.
EFI_INVALID_PARAMETER	The PIRQ number invalid.

EFI_LEGACY_INTERRUPT_PROTOCOL.WritePirq()

Summary

Writes data to the specified PIRQ register.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_INTERRUPT_WRITE_PIRQ) (
  IN  EFI_LEGACY_INTERRUPT_PROTOCOL  *This,
  IN  UINT8                          PirqNumber,
  IN  UINT8                          PirqData
  )
```

Parameters

This
Indicates the **EFI_LEGACY_INTERRUPT_PROTOCOL** instance.

PirqNumber
PIRQ A = 0, PIRQB = 1, and so on.

PirqData
IRQ assigned to this PIRQ

Description

This function writes the indicated PIRQ register with the requested data.

Status Codes Returned

EFI_SUCCESS	The PIRQ was programmed.
EFI_INVALID_PARAMETER	The PIRQ is not valid.

3.3 Compatibility16 Code

3.3.1 Compatibility16 Code

The runtime Compatibility16 code (traditional 16-bit runtime code) is loaded as a binary file during the installation of **EFI_LEGACY_BIOS_PROTOCOL**. **EFI_LEGACY_BIOS_PLATFORM_PROTOCOL.GetSystemRom()** is invoked, which finds the appropriate binary file. The GUID referring to this binary is IBV specific and may be specific for an OEM supported by the IBV.

GUID

IBV or OEM specific

3.3.2 Legacy BIOS Interface

3.3.2.1 Compatibility16 Table

EFI_COMPATIBILITY16_TABLE

Summary

There is a table located within the traditional BIOS in either the 0xF000:xxxx or 0xE000:xxxx physical address range. It is located on a 16-byte boundary and provides the physical address of the entry point for the Compatibility16 functions. These functions provide the platform-specific information that is required by the generic EfiCompatibility code. The functions are invoked via thunking by using **EFI_LEGACY_BIOS_PROTOCOL.FarCall86()** with the 32-bit physical entry point defined below.

Prototype

```
typedef struct {
  UINT32          Signature;
  UINT8           TableChecksum;
  UINT8           TableLength;
  UINT8           EfiMajorRevision;
  UINT8           EfiMinorRevision;
  UINT8           TableMajorRevision;
  UINT8           TableMinorRevision;
  UINT16          Reserved;
  UINT16          Compatibility16CallSegment;
  UINT16          Compatibility16CallOffset;
  UINT16          PnPInstallationCheckSegment;
```

```
    UINT16          PnPInstallationCheckOffset;
    UINT32          EfiSystemTable;
    UINT32          OemIdStringPointer;
    UINT32          AcpiRsdPtrPointer;
    UINT16          OemRevision;
    UINT32          E820Pointer;
    UINT32          E820Length;
    UINT32          IrqRoutingTablePointer;
    UINT32          IrqRoutingTableLength;
    UINT32          MpTablePtr;
    UINT32          MpTableLength;
    UINT16          OemIntSegment;
    UINT16          OemIntOffset;
    UINT16          Oem32Segment;
    UINT16          Oem32Offset;
    UINT16          Oem16Segment;
    UINT16          Oem16Offset;
    UINT16          TpmSegment;
    UINT16          TpmOffset;
    UINT32          IbvPointer;
    UINT32          PciExpressBase;
    UINT8           LastPciBus;
  } EFI_COMPATIBILITY16_TABLE;
```

Parameters

Signature

The string “$EFI” denotes the start of the EfiCompatibility table. Byte 0 is “I,” byte 1 is “F,” byte 2 is “E,” and byte 3 is “$” and is normally accessed as a DWORD or UINT32.

TableChecksum

The value required such that byte checksum of *TableLength* equals zero.

TableLength

The length of this table.

EfiMajorRevision

The major EFI revision for which this table was generated.

EfiMinorRevision

The minor EFI revision for which this table was generated.

TableMajorRevision

The major revision of this table.

TableMinorRevision

The minor revision of this table.

Reserved

Reserved for future usage.

Compatibility16CallSegment

The segment of the entry point within the traditional BIOS for Compatibility16 functions.

Compatibility16CallOffset

The offset of the entry point within the traditional BIOS for Compatibility16 functions.

PnPInstallationCheckSegment

The segment of the entry point within the traditional BIOS for EfiCompatibility to invoke the PnP installation check.

PnPInstallationCheckOffset

The Offset of the entry point within the traditional BIOS for EfiCompatibility to invoke the PnP installation check.

EfiSystemTable

Pointer to EFI system resources table. EFI system resources table is of the type **EFI_SYSTEM_TABLE** defined in the *Intel® Platform Innovation Framework for EFI Driver Execution Environment Core Interface Specification* (DXE CIS).

OemIdStringPointer

The address of an OEM-provided identifier string. The string is null terminated.

AcpiRsdPtrPointer

The 32-bit physical address where ACPI RSD PTR is stored within the traditional BIOS. The remained of the ACPI tables are located at their EFI addresses. The size reserved is the maximum for ACPI 2.0. The EfiCompatibility will fill in the ACPI RSD PTR with either the ACPI 1.0b or 2.0 values.

OemRevision

The OEM revision number. Usage is undefined but provided for OEM module usage.

E820Pointer

The 32-bit physical address where INT15 E820 data is stored within the traditional BIOS. The EfiCompatibility code will fill in the E820Pointer value and copy the data to the indicated area.

E820Length

The length of the E820 data and is filled in by the EfiCompatibility code.

IrqRoutingTablePointer

The 32-bit physical address where the **$PIR** table is stored in the traditional BIOS. The EfiCompatibility code will fill in the *IrqRoutingTablePointer* value and copy the data to the indicated area.

IrqRoutingTableLength

The length of the **$PIR** table and is filled in by the EfiCompatibility code.

MpTablePtr

The 32-bit physical address where the MP table is stored in the traditional BIOS. The EfiCompatibility code will fill in the *MpTablePtr* value and copy the data to the indicated area.

MpTableLength

The length of the MP table and is filled in by the EfiCompatibility code.

Oemint15Segment

The segment of the OEM-specific INT table/code.

OemInt15Offset

The offset of the OEM-specific INT table/code.

Oem32Segment

The segment of the OEM-specific 32-bit table/code.

Oem32Offset

The offset of the OEM-specific 32-bit table/code.

Oem16Segment

The segment of the OEM-specific 16-bit table/code.

Oem16Offset

The offset of the OEM-specific 16-bit table/code.

TpmSegment

The segment of the TPM binary passed to 16-bit CSM.

TpmOffset

The offset of the TPM binary passed to 16-bit CSM.

IbvPointer

A pointer to a string identifying the independent BIOS vendor.

PciExpressBase

This field is **NULL** for all systems not supporting PCI Express. This field is the base value of the start of the PCI Express memory-mapped configuration registers and must be filled in prior to EfiCompatibility code issuing the Compatibility16 function **Compatibility16InitializeYourself()**. **Compatibility16InitializeYourself()** is defined in Compatability16 Functions.

LastpciBus

Maximum PCI bus number assigned.

Note: *The* E820Pointer, IrqRoutingTablePointer, *and* MpTablePtr *values are generated by calling the* ***Compatibility16GetTableAddress()*** *function and converted to 32-bit physical pointers.*

3.3.3 Compatibility16 Functions

These functions are accessed by the EfiCompatibility code using the **EFI_LEGACY_BIOS_PROTOCOL.FarCall86()** call with the segment:offset equivalent of the 32-bit physical entry point for legacy EFI services.

Note that the **EFI_COMPATIBILITY_FUNCTIONS** are for IA-32. Unused registers on input and on output are undefined and not guaranteed to be preserved. Equivalents for the Itanium® processor family are not defined at this time.

Note: *Register AX denotes the function that is requested and the rest of the registers are function dependant.*

Functions 0x0000–0x7FFF are standard Compatibility16 functions.

Functions 0x8000–0xFFFF are OEM-defined Compatibility16 functions and outside the scope of this document.

3.3.3.1 EFI Compatibility Functions

EFI_COMPATIBILITY_FUNCTIONS

Summary

Functions to communicate between the EfiCompatibility and Compatability16 code.

Prototype

```
typedef enum {
  Compatibility16InitializeYourself      0000,
  Compatibility16UpdateBbs               0001,
  Compatibility16PrepareToBoot           0002,
  Compatibility16Boot                    0003,
  Compatibility16RetrieveLastBootDevice  0004,
  Compatibility16DispatchOprom           0005,
  Compatibility16GetTableAddress         0006,
  Compatibility16SetKeyboardLeds         0007,
  Compatibility16InstallPciHandler       0008,
} EFI_COMPATIBILITY_FUNCTIONS;
```

Parameters

`Compatibility16InitializeYourself`

Causes the Compatibility16 code to do any internal initialization required. See the **`Compatibility16InitializeYourself()`** function description.

`Compatibility16UpdateBbs`

Causes the Compatibility16 BIOS to perform any drive number translations to match the boot sequence. See the **`Compatibility16UpdateBbs()`** function description.

`Compatibility16PrepareToBoot`

Allows the Compatibility16 code to perform any final actions before booting. See the **`Compatibility16PrepareToBoot()`** function description.

`Compatibility16Boot`

Causes the Compatibility16 BIOS to boot. See the **`Compatibility16Boot()`** function description.

`Compatibility16RetrieveLastBootDevice`

Allows the Compatibility16 code to get the last device from which a boot was attempted. See the **`Compatibility16RetrieveLastBootDevice()`** function description.

`Compatibility16DispatchOprom`

Allows the Compatibility16 code rehook INT13, INT18, and/or INT19 after dispatching a legacy OpROM. See the **`Compatibility16DispatchOprom()`** function description.

`Compatibility16GetTableAddress`

Finds a free area in the 0xFxxxx or 0xExxxx region of the specified length and returns the address of that region. See the **`Compatibility16GetTableAddress()`** function description.

`Compatibility16SetKeyboardLeds`

Enables the EfiCompatibility module to do any nonstandard processing of keyboard LEDs or state. See the **`Compatibility16SetKeyboardLeds()`** function description.

`Compatibility16InstallPciHandler`

Enables the EfiCompatibility module to install an interrupt handler for PCI mass media devices that do not have an OpROM associated with them. See the **`Compatibility16InstallPciHandler()`** function description.

Compatibility16InitializeYourself()

Summary

Causes the Compatibility16 code to do any internal initialization required. The **EFI_TO_COMPATIBILITY16_INIT_TABLE** pointer is passed into this function.

Input Registers

AX = **Compatibility16InitializeYourself**

ES:BX = Pointer to **EFI_TO_COMPATIBILITY16_INIT_TABLE**

Output Registers

AX = Return Status codes

Related Definitions

```
//****************************************************************
// EFI_TO_COMPATIBILITY16_INIT_TABLE
//**************************************************************
typedef struct {
  UINT32          BiosLessThan1MB;
  UINT32          HiPmmMemory;
  UINT32          HIPmmMemorySizeInBytes;
  UINT16          ReverseThunkCallSegment;
  UINT16          ReverseThunkCallOffset;
  UINT32          NumberE820Entries;
  UINT32          OsMemorybove1Mb;
  UINT32          ThunkStart;
  UINT32          ThunkSizeInBytes;
  UINT32          LowPmmMemory;
  UINT32          LowPmmMemorySizeInBytes;
} EFI_TO_COMPATIBILITY16_INIT_TABLE;
```

BiosLessThan1MB

Starting address of low memory block (below 1MB) that can be used for PMM. The end address is assumed to be 0x9FFFF. This field is maintained for compatibility with previous versions of the specification and the CSM16 should not use this field.

The CSM16 should use *LowPmmMemory* and *LowPmmMemorySizeInBytes* fields for the low memory that can be used for PMM.

HiPmmMemory
Starting address of the high memory block.

HiPmmMemorySizeInBytes
Length of high memory block.

ReverseThunkCallSegment
The segment of the reverse thunk call code.

ReverseThunkCallOffset
The offset of the reverse thunk call code.

Number820Entries
The number of E820 entries copied to the Compatibility16 BIOS.

OsMemoryAbove1Mb
The amount of usable memory above 1 MB, e.g., E820 type 1 memory.

ThunkStart
The start of thunk code in main memory. Memory cannot be used by BIOS or PMM.

ThunkSizeInBytes
The size of the thunk code.

LowPmmMemory
Starting address of memory under 1 MB.

LowPmmMemorySizeInBytes
Length of low Memory block.

Note: *The address of the* `ReverseThunkCall` *code is provided in case the Compatibility16 code needs to invoke a Compatibility16 function. It is not used to return from this function or any other traditional BIOS interface function. These functions simply do a far return.*

Note: *CSM16 must handle cases where the PMM pointers are NULL. That indicates that PMM is not supported for that range. If both pointers are NULL then PMM is not supported. This covers cases where no add-in cards are supported and/or memory given to EFI.*

Note: *CSM16 must initialize the PMM regions to zero prior to usage by OPROMS. CSM16 should not assume the CSM32 has zeroed out the regions.*

Note: *CSM16 must monitor for EBDA size increase after OPROM is initialized and adjust PMM below 1MB, if required.*

Status Codes Returned

EFI_SUCCESS	0x0000

Compatibility16UpdateBbs()

Summary

Causes the Compatibility16 BIOS to perform any drive number translations to match the boot sequence.

Input Registers

AX = **`Compatibility16UpdateBbs`**

ES:BX = Pointer to **`EFI_TO_COMPATIBILITY16_BOOT_TABLE`**

Output Registers

AX = Returned status codes

Status Codes Returned

EFI_SUCCESS	0x0000

Compatibility16PrepareToBoot()

Summary

Allows the Compatibility16 code to perform any final actions before booting. The Compatibility16 code is read/write.

Input Registers

AX = **Compatibility16PrepareToBoot**

ES:BX = Pointer to **EFI_TO_COMPATIBILITY16_BOOT_TABLE** structure

Output Registers

AX = Returned status codes

Related Definitions

The following data types and structures are defined in this section. These definitions in turn may contain other data type and structure definitions that are not included in this list.

- **EFI_TO_COMPATIBILITY16_BOOT_TABLE**
- **DEVICE_PRODUCER_DATA_HEADER**
- **HDD_INFO**
- **BBS_TABLE**
- **BBS_STATUS_FLAGS**
- **SMM_TABLE**
- **UD_TABLE**
- **UDC_ATTRIBUTES**

```
//****************************************************************
// EFI_TO_COMPATIBILITY16_BOOT_TABLE
//****************************************************************
typedef struct {
  UINT16              MajorVersion;
  UINT16              MinorVersion;
  UINT32              AcpiTable;          // 4 GB range
  UINT32              SmbiosTable;        // 4 GB range
  UINT32              SmbiosTableLength;
```

```
    //
    // Legacy SIO state
    //

    DEVICE_PRODUCER_DATA_HEADER    SioData;
    UINT16                         DevicePathType;
    UINT16                         PciIrqMask;
    UINT32                         NumberE820Entries;

    //
    // Controller & Drive Identify[2] per controller information
    //

    HDD_INFO            HddInfo[MAX_IDE_CONTROLLER];
    UINT32              NumberBbsEntries;
UINT32              BbsTable;
UINT32              SmmTable;
    UINT32              OsMemoryAbove1Mb;
UINT32              UnconventionalDeviceTable;
  } EFI_TO_COMPATIBILITY16_BOOT_TABLE;
```

MajorVersion

The EfiCompatibility major version number.

MinorVersion

The EfiCompatibility minor version number.

AcpiTable

Location of the RSDT ACPI table.

SmbiosTable

Location of the SMBIOS table in EFI memory.

SioData

Standard traditional device information. Type **DEVICE_PRODUCER_DATA_HEADER** is defined below.

DevicePathType

The default boot type. Following are the defined values:

1. FD = Floppy
2. HD = Hard Disk
3. CDROM = CD-ROM
4. PCMCIA = PCMCIA
5. USB = USB
6. NET = Networks
7. BEV = BBS BEV devices

PciIrqMask

Mask of which IRQs have been assigned to PCI.

NumberE820Entries

Number of E820 entries. The number can change from the **Compatibility16InitializeYourself()** function.

HddInfo

Hard disk drive information, including raw Identify Drive data. Type **HDD_INFO** is defined below.

NumberBbsEntries

Number of entries in the BBS table

BbsTable

Pointer to the BBS table. Type **BBS_TABLE** is defined below.

SmmTable

Pointer to the SMM table. Type *SMM_TABLE* is defined below.

OsMemoryAbove1Mb

The amount of usable memory above 1 MB, i.e. E820 type 1 memory. This value can differ from the value in **EFI_TO_COMPATIBILITY16_INIT_TABLE** as more memory may have been discovered.

UnconventionalDeviceTable

Information to boot off an unconventional device like a PARTIES partition. Type **UD_TABLE** is defined below.

```
//*****************************************************

// DEVICE_PRODUCER_DATA_HEADER

//*****************************************************

typedef struct {

  DEVICE_PRODUCER_SERIAL    Serial[4];

  DEVICE_PRODUCER_PARALLEL  Parallel[3];
```

```
  DEVICE_PRODUCER_FLOPPY     Floppy;
  UINT8                      MousePresent;
  LEGACY_DEVICE_FLAGS        Flags;
} DEVICE_PRODUCER_DATA_HEADER;
```

Serial
Data for serial port x. Type **DEVICE_PRODUCER_SERIAL** is defined below.

Parallel
Data for parallel port x. Type **DEVICE_PRODUCER_PARALLEL** is defined below.

Floppy
Data for floppy. Type **DEVICE_PRODUCER_FLOPPY** is defined below.

MousePresent
Flag to indicate if mouse is present.

Flags
Miscellaneous Boolean state information passed to CSM. Type **LEGACY_DEVICE_FLAGS** is defined below.

```
//******************************************************
// DEVICE_PRODUCER_SERIAL
//******************************************************
typedef struct {
  UINT16          Address;
  UINT8           Irq;
  SERIAL_MODE     Mode;
} DEVICE_PRODUCER_SERIAL;
```

Address
I/O address assigned to the serial port

Irq
IRQ assigned to the serial port.

Mode
Mode of serial port. Values are defined below.

```
//*****************************************************
// Serial Mode values
//*****************************************************
#define DEVICE_SERIAL_MODE_NORMAL              0x00
#define DEVICE_SERIAL_MODE_IRDA                0x01
#define DEVICE_SERIAL_MODE_ASK_IR              0x02
#define DEVICE_SERIAL_MODE_DUPLEX_HALF         0x00
#define DEVICE_SERIAL_MODE_DUPLEX_FULL         0x10


//*****************************************************
// DEVICE_PRODUCER_PARALLEL
//*****************************************************

typedef struct {
  UINT16          Address;
  UINT8           Irq;
  UINT8           Dma;
  PARALLEL_MODE   Mode;
} DEVICE_PRODUCER_PARALLEL;
```

Address

I/O address assigned to the parallel port

Irq

IRQ assigned to the parallel port.

Dma

DMA assigned to the parallel port.

Mode

Mode of the parallel port. Values are defined below.

```
//*****************************************************
```

```
// Parallel Mode values
//*******************************************************
#define DEVICE_PARALLEL_MODE_MODE_OUTPUT_ONLY    0x00
#define DEVICE_PARALLEL_MODE_MODE_BIDIRECTIONAL  0x01
#define DEVICE_PARALLEL_MODE_MODE_EPP            0x02
#define DEVICE_PARALLEL_MODE_MODE_ECP            0x03
```

```
//*******************************************************
// DEVICE_PRODUCER_FLOPPY
//*******************************************************
typedef struct {
  UINT16          Address;
  UINT8           Irq;
  UINT8           Dma;
  UINT8           NumberOfFloppy;
} DEVICE_PRODUCER_FLOPPY;
```

Address
: I/O address assigned to the floppy

Irq
: IRQ assigned to the floppy.

Dma
: DMA assigned to the floppy.

NumberOfFloppy
: Number of floppies in the system.

```
//*******************************************************
// LEGACY_DEVICE_FLAGS
//*******************************************************
```

```
typedef struct {
  UINT32          A20Kybd:1;
  UINT32          A20Port92:1
  UINT32          Reserved:30;
} LEGACY_DEVICE_FLAGS;
```

A20Kybd
A20 controller by keyboard controller.

A20Port92
A20 controlled by port 0x92.

Reserved
Reserved for future usage.

Note: *`A20Kybd` and `A20Port92` are not mutually exclusive.*

```
//***************************************************
// HDD_INFO
//***************************************************

typedef struct {
  UINT16                    Status;
  UINT32                    Bus;
  UINT32                    Device;
  UINT32                    Function;
  UINT16                    CommandBaseAddress;
  UINT16                    ControlBaseAddress;
  UINT16                    BusMasterAddress;
  UINT8                     HddIrq;
  ATAPI_IDENTIFY            IdentifyDrive[2];
} HDD_INFO;
```

Status

Status of IDE device. Values are defined below. There is one **HDD_INFO** structure per IDE controller. The *IdentifyDrive* is per drive. Index 0 is master and index 1 is slave.

Bus

PCI bus of IDE controller.

Device

PCI device of IDE controller.

Function

PCI function of IDE controller.

CommandBaseAddress

Command ports base address.

ControlBaseAddress

Control ports base address.

BusMasterAddress

Bus master address

IdentifyDrive

Data that identifies the drive data, one per possible attached drive. Type **ATAPI_IDENTIFY** is defined below.

```
//***************************************************
// Status values
//***************************************************

#define HDD_PRIMARY                0x01
#define HDD_SECONDARY              0x02
#define HDD_MASTER_ATAPI           0x04
#define HDD_SLAVE_ATAPI            0x08
#define HDD_MASTER_ATAPI_ZIPDISK   0x10
#define HDD_MASTER_IDE             0x20
#define HDD_SLAVE_IDE              0x40
#define HDD_SLAVE_ATAPI_ZIPDISK    0x80
```

```
//**************************************************
// ATAPI_IDENTIFY
//**************************************************

typedef struct {
  UINT16    Raw[256];
} ATAPI_IDENTIFY;
```

Raw

Raw data from the IDE *IdentifyDrive* command.

```
//*****************************************************
// BBS_TABLE
//*****************************************************

typedef struct {
  UINT16              BootPriority;
  UINT32              Bus;
  UINT32              Device;
  UINT32              Function;
  UINT8               Class;
  UINT8               SubClass;
  UINT16              MfgStringOffset;
  UINT16              MfgStringSegment;
  UINT16              DeviceType;
  BBS_STATUS_FLAGS    StatusFlags;
  UINT16              BootHandlerOffset;
  UINT16              BootHandlerSegment;
  UINT16              DescStringOffset;
```

```
    UINT16              DescStringSegment;
    UINT32              InitPerReserved;
    UINT32              AdditionalIrq13Handler;
    UINT32              AdditionalIrq18Handler;
    UINT32              AdditionalIrq19Handler;
    UINT32              AdditionalIrq40Handler;
    UINT8               AssignedDriveNumber;
    UINT32              AdditionalIrq41Handler;
    UINT32              AdditionalIrq46Handler;
    UINT32              IBV1;
    UINT32              IBV2;
  } BBS_TABLE;
```

BootPriority
The boot priority for this boot device. Values are defined below.

Bus
The PCI bus for this boot device.

Device
The PCI device for this boot device.

Function
The PCI function for the boot device.

Class
The PCI class for this boot device..

SubClass
The PCI Subclass for this boot device.

MfgString
Segment:offset address of an ASCIIZ description string describing the manufacturer.

DeviceType
BBS device type. BBS device types are defined below.

StatusFlags
Status of this boot device. Type **BBS_STATUS_FLAGS** is defined below.

BootHandler
Segment:Offset address of boot loader for IPL devices or install INT13 handler for BCV devices.

DescString

Segment:offset address of an ASCIIZ description string describing this device.

InitPerReserved

Reserved.

AdditionalIrq??Handler

The use of these fields is IBV dependent. They can be used to flag that an OpROM has hooked the specified IRQ. The OpROM may be BBS compliant as some SCSI BBS-compliant OpROMs also hook IRQ vectors in order to run their BIOS Setup.

```
//****************************************************
// BootPriority values
//****************************************************
#define BBS_DO_NOT_BOOT_FROM        0xFFFC
#define BBS_LOWEST_PRIORITY         0xFFFD
#define BBS_UNPRIORITIZED_ENTRY     0xFFFE
#define BBS_IGNORE_ENTRY            0xFFFF
```

Table 11 gives a description of the above fields.

Table 11 BBS Fields

BBS_DO_NOT_BOOT_FROM	Removes a device from the boot list but still allows it to be enumerated as a valid device under MS-DOS*.
BBS_LOWEST_PRIORITY	Forces the device to be the last boot device.
BBS_UNPRIORITIZED_ENTRY	Value that is placed in the **BBS_TABLE.***BootPriority* field before priority has been assigned but that indicates it is valid entry. Other values indicate the priority, with 0x0000 being the highest priority.
BBS_IGNORE_ENTRY	When placed in the **BBS_TABLE.***BootPriority* field, indicates that the entry is to be skipped.

```
//****************************************************
// DeviceType values
//****************************************************
#define    BBS_FLOPPY         0x01
#define    BBS_HARDDISK       0x02
#define    BBS_CDROM          0x03
#define    BBS_PCMCIA         0x04
#define    BBS_USB            0x05
#define    BBS_EMBED_NETWORK  0x06
#define    BBS_BEV_DEVICE     0x80
#define    BBS_UNKNOWN        0xff


//****************************************************
// BBS_STATUS_FLAGS
//****************************************************

typedef struct {
  UINT16                  OldPosition : 4;
  UINT16                  Reserved1   : 4;
  UINT16                  Enabled     : 1;
  UINT16                  Failed      : 1;
  UINT16                  MediaPresent: 2;
  UINT16                  Reserved2   : 4;
} BBS_STATUS_FLAGS ;
```

OldPosition

Prior priority.

Reserved1

Reserved for future use.

Enabled

If 0, ignore this entry.

Failed

0 = Not known if boot failure occurred.

1 = Boot attempted failed.

MediaPresent

State of media present.

00 = No bootable media is present in the device.

01 = Unknown if a bootable media present.

10 = Media is present and appears bootable.

11 = Reserved.

Reserved2

Reserved for future use.

```
//****************************************************
// SMM_TABLE
//****************************************************
//
// SMM Table definitions
// SMM table has a header that provides the number of entries.
// Following the header is a variable length amount of data.
//

typedef struct {
  UINT16            NumSmmEntries;
  SMM_ENTRY         SmmEntry;
} SMM_TABLE;
```

NumSmmEntries

Number of entries represented by *SmmEntry*.

SmmEntry

One entry per function. Type **SMM_ENTRY** is defined below.

```
//****************************************************
// SMM_ENTRY
//****************************************************
typedef struct {
   SMM_ATTRIBUTES     SmmAttributes;
   SMM_FUNCTION       SmmFunction;
   UINTx              SmmPort;
   UINTx              SmmData;
} SMM_ENTRY;
```

SmmAttributes
: Describes the access mechanism, *SmmPort*, and *SmmData* sizes. Type **SMM_ATTRIBUTES** is defined below.

SmmFunction
: Function Soft SMI is to perform. Type **SMM_FUNCTION** is defined below.

SmmPort
: *SmmPort* size depends upon *SmmAttributes* and ranges from 1 bytes to 8 bytes

SmmData
: *SmmData* size depends upon *SmmAttributes* and ranges from 1 bytes to 8 bytes

Note: *The SmmPort and SmmData are packed in order to present the smallest footprint for the CSM16. Typically the user will set a pointer to the SmmPort and then use a structure like the one below to access the port and data.*

```
typedef struct {
   UINT8             SmmPort;
   UINT8             SmmData;
} P8D8;
```

```
//****************************************************
// SMM_ATTRIBUTES
```

```
//*******************************************************
typedef struct {
   UINT16          Type                : 3;
   UINT16          PortGranularity     : 3;
   UINT16          DataGranularity     : 3;
   UINT16          Reserved            : 7;
} SMM_ATTRIBUTES;
```

Type

Access mechanism used to generate the soft SMI. Defined types are below. The other values are reserved for future usage.

PortGranularity

Size of "port" in bits. Defined values are below.

DataGranularity

Size of data in bits. Defined values are below.

Reserved

Reserved for future use.

```
//*******************************************************
// Type values
//*******************************************************
#define STANDARD_IO      0x00
#define STANDARD_MEMORY  0x01

//*******************************************************
// PortGranularity values
//*******************************************************
#define  PORT_SIZE_8     0x00
#define  PORT_SIZE_16    0x01
#define  PORT_SIZE_32    0x02
#define  PORT_SIZE_64    0x03
```

```
//****************************************************
// DataGranularity values
//****************************************************
#define  DATA_SIZE_8     0x00
#define  DATA_SIZE_16    0x01
#define  DATA_SIZE_32    0x02
#define  DATA_SIZE_64    0x03
```

```
//****************************************************
// SMM_FUNCTION
//****************************************************
typedef struct {
   UINT16            Function          : 15;
   UINT16            Owner             : 1;
} SMM_FUNCTION;
```

Function

Function this Soft SMI is to initiate. Defined functions are below.

Owner

The definer of the function. Defined owners are below.

```
//****************************************************
// Function values
//****************************************************
#define  INT15_D042          0x0000
#define  GET_USB_BOOT_INFO   0x0001
#define  DMI_PNP_50_57       0x0002
```

Table 12 gives a description of the fields in the above definition.

Table 12 Function Value Descriptions

INT15_D042	System Configuration Data functions accessed via INT15 AX=0xD042.
GET_USB_BOOT_INFO	Retrieves USB boot device information for integration with BBS. The other values are reserved for future use.
DMI_PNP_50_57	Process the DMI Plug and Play functions 0x50 through 0x57 via SMM code.

```
//*******************************************************
// Owner values
//*******************************************************
#define  STANDARD_OWNER       0x0
#define  OEM_OWNER            0x1
```

Table 13 givesa description of the fields in the above definition.

Table 13 Owner Value Descriptions

STANDARD_OWNER	This document has defined the function.
OEM_OWNER	An agent, other than this document, has defined the function.

```
//*******************************************************
// UD_TABLE
//*******************************************************
typedef struct {
   UDC_ATTRIBUTES     Attributes;
   UINT8              DeviceNumber;
   UINT8              BbsTableEntryNumberForParentDevice;
   UINT8              BbsTableEntryNumberForBoot;
```

```
    UINT8          BbtTableEntryNumberForHddDiag;
    UINT8          BeerData[128];
    UINT8          ServiceAreaData[64];
} UD_TABLE;
```

Attributes

This field contains the bit-mapped attributes of the PARTIES information. Type **UDC_ATTRIBUTES** is defined below.

DeviceNumber

This field contains the zero-based device on which the selected *ServiceDataArea* is present. It is 0 for master and 1 for the slave device.

BbsTableEntryNumberForParentDevice

This field contains the zero-based index into the *BbsTable* for the parent device. This index allows the user to reference the parent device information such as PCI bus, device function.

BbsTableEntryNumberForBoot

This field contains the zero-based index into the *BbsTable* for the boot entry.

BbsTableEntryNumberForHddDiag

This field contains the zero-based index into the *BbsTable* for the HDD diagnostics entry.

BeerData

The raw Beer data.

ServiceAreaData

The raw data of selected service area.

```
//*******************************************************
// UDC_ATTRIBUTES
//*******************************************************
typedef struct {
   UINT8          DirectoryServiceValidity      : 1;
   UINT8          RabcaUsedFlag      : 1;
   UINT8          ExecuteHddDiagnosticsFlag     : 1;
   UINT8          Reserved     : 5;
} UDC_ATTRIBUTES;
```

DirectoryServiceValidity
This bit set indicates that the *ServiceAreaData* is valid.

RacbaUsedFlag
This bit set indicates to use the Reserve Area Boot Code Address (RACBA) only if *DirectoryServiceValidity* is 0.

ExecuteHddDiagnosticsFlag
This bit set indicates to execute hard disk diagnostics.

Reserved
Reserved for future use. Set to 0.

Status Codes Returned

EFI_SUCCESS	0x0000

Compatibility16Boot()

Summary

Causes the Compatibility16 BIOS to boot. The Compatibility16 code is Read/Only.

Input Registers

AX = **Compatibility16Boot**

Output Registers

AX = Returned status codes

Related Definitions

```
typedef struct {
} EFI_COMPATIBILITY16_BOOT;
```

Status Codes Returned

EFI_SUCCESS	0x0000
EFI_TBD	0x8000 – The master boot record is missing or corrupted.

Compatibility16RetrieveLastBootDevice()

Summary

Allows the Compatibility16 code to get the last device from which a boot was attempted. This is stored in CMOS and is the priority number of the last attempted boot device.

Input Registers

AX = **`Compatibility16RetrieveLastBootDevice`**

Output Registers

AX = Returned status codes

BX = Priority number of the boot device.

Status Codes Returned

EFI_SUCCESS	0x0000
EFI_ABORTED	0x8015

Compatibility16DispatchOprom()

Summary

Allows the Compatibility16 code rehook INT13, INT18, and/or INT19 after dispatching a legacy OpROM.

Input Registers

AX = **Compatibility16DispatchOprom**

ES:BX = Pointer to **EFI_DISPATCH_OPROM_TABLE**

Output Registers

AX = Returned status codes

BX = Number of non-BBS-compliant devices found. Equals 0 if BBS compliant.

Related Definitions

```
//**********************************************************
// EFI_DISPATCH_OPROM_TABLE
//**********************************************************
typedef struct {
  UINT16          PnpInstallationCheckSegment;
  UINT16          PnpInstallationCheckOffset;
  UINT16          OpromSegment;
  UINT8           PciBus;
  UINT8           PciDeviceFunction
  UINT8           NumberBbbsEntries;
  UINT32          BbsTablePointer;
  UINT16          OpromDestinationSegment;
} EFI_DISPATCH_OPROM_TABLE;
```

PnpInstallationCheckSegment/Offset
Pointer to the *PnpInstallationCheck* data structure.

OpromDestinationSegment
The segment where the OpROM can be relocated to. If this value is 0x0000, this means that the relocation of this run time code is not supported.

OpromSegment
The segment where the OpROM was placed. Offset is assumed to be 3.

PciBus

The PCI bus.

PciDeviceFunction

The PCI device * 0x08 | PCI function.

NumberBbsEntries

The number of valid BBS table entries upon entry and exit. The IBV code may increase this number, if BBS-compliant devices also hook INTs in order to force the OpROM BIOS Setup to be executed.

BbsTable

Pointer to the BBS table.

Status Codes Returned

EFI_SUCCESS	0x0000

Compatibility16GetTableAddress()

Summary

Finds a free area in the 0xFxxxx or 0xExxxx region of the specified length and returns the address of that region.

Input Registers

AX = **Compatibility16GetTableAddress**

BX = Allocation region

00 = Allocate from either 0xE0000 or 0xF0000 64 KB blocks.

Bit 0 = 1 Allocate from 0xF0000 64 KB block

Bit 1 = 1 Allocate from 0xE0000 64 KB block

CX = Requested length in bytes.

DX = Required address alignment. Bit mapped. First non-zero bit from the right is the alignment.

Output Registers

AX = Returned status codes

DS:BX = Address of the region

Status Codes Returned

EFI_SUCCESS	0x0000
EFI_OUT_OF_RESOURCES	0x8009

Compatibility16SetKeyboardLeds()

Summary

Enables the EfiCompatibility module to do any nonstandard processing of keyboard LEDs or state.

Input Registers

AX = **Compatibility16SetKeyboardLeds**

CL = LED status.

Bit 0 – Scroll Lock   0 = Off

Bit 1 – Num Lock

Bit 2 – Caps Lock

Output Registers

AX = Returned status codes

Status Codes Returned

EFI_SUCCESS	0x0000

Compatibility16InstallPciHandler()

Summary

Enables the EfiCompatibility module to install an interrupt handler for PCI mass media devices that do not have an OpROM associated with them. An example is SATA.

Input Registers

AX = **Compatibility16InstallPciHandler**

ES:BX = Pointer to **EFI_LEGACY_INSTALL_PCI_HANDLER** structure

Output Registers

AX = Returned status codes

Related Definitions

```
//****************************************************
// EFI_LEGACY_INSTALL_PCI_HANDLER
//****************************************************
typedef struct {
  UINT8             PciBus;
  UINT8             PciDeviceFun;
  UINT8             PciSegment;
  UINT8             PciClass;
  UINT8             PciSubclass;
  UINT8             PciInterface;
  //
  // Primary section
  //
  UINT8             PrimaryIrq;
  UINT8             PrimaryReserved;
  UINT16            PrimaryControl;
  UINT16            PrimaryBase;
  UINT16            PrimaryBusMaster;

  //
```

```
    // Secondary section
    //
    UINT8                   SecondaryIrq;
    UINT8                   SecondaryReserved;
    UINT16                  SecondaryControl;
    UINT16                  SecondaryBase;
    UINT16                  SecondaryBusMaster;
  } EFI_LEGACY_INSTALL_PCI_HANDLER;
```

PciBus
The PCI bus of the device.

PciDeviceFun
The PCI device in bits 7:3 and function in bits 2:0.

PciSegment
The PCI segment of the device.

PciClass
The PCI class code of the device.

PciSubclass
The PCI subclass code of the device.

PciInterface
The PCI interface code of the device.

PrimaryIrq
The primary device IRQ.

PrimaryReserved
Reserved.

PrimaryControl
The primary device control I/O base.

PrimaryBase
The primary device I/O base.

PrimaryBusMaster
The primary device bus master I/O base.

SecondaryIrq
The secondary device IRQ.

SecondaryReserved
Reserved.

SecondaryControl

The secondary device control I/O base.

SecondaryBase

The secondary device I/O base.

SecondaryBusMaster

The secondary device bus master I/O base.

Status Codes Returned

EFI_SUCCESS	0x0000

3.3.3.2 Legacy Soft SMI

Summary

SMM code is provided from the same IBV as the Compatibility16 and the Legacy BIOS Platform Protocol code. The soft SMI structures in **`SMM_TABLE`** (defined in **`Compatibility16PrepareToBoot()`**) are meant to establish a common standard but are not required to be implemented by the IBV. If the structures are used, then the recommended interface between the SMM code and Compatibility16 code is defined below.

Input Registers

EAX, AX, AL = **`SmmTable.SmmEntry.SmmData`** (32-bit, 16-bit, or 8-bit)

EDX, DX, DL = **`SmmTable.SmmEntry.SmmPort`** (32-bit, 16-bit, or 8-bit)

ESI = Pointer to **`EFI_DWORD_REGS`**

Related Definitions

EFI_DWORD_REGS is defined in the **`EFI_LEGACY_BIOS_PROTOCOL.Int86()`** function.

4
Example Code

4.1 Example of a Dummy EFI SMM Child Driver

User defined areas are highlighted like this in yellow.

```
/*++

Module Name:

    UnitTestChild.c

Abstract:

    This is a generic template for a child of the IchSmm driver.


--*/

#include "Efi.h"
#include "EfiRuntimeLib.h"

#include "GetFvImage.h"

#include EFI_PROTOCOL_CONSUMER(SmmBase)
#include EFI_PROTOCOL_CONSUMER(FirmwareVolume)
#include EFI_PROTOCOL_CONSUMER(SmmSwDispatch)

EFI_SMM_BASE_PROTOCOL        *mSmmBase;
EFI_SMM_SYSTEM_TABLE         *mSmst;
EFI_SMM_SW_DISPATCH_PROTOCOL *mSwDispatch;

// GUID for the FV file that this source file gets compiled into
EFI_GUID  mChildFileGuid = { your guid here };

////////////////////////////////////////////
// Callback function prototypes

VOID
SwCallback (
  IN  EFI_HANDLE                    DispatchHandle,
  IN  EFI_SMM_SW_DISPATCH_CONTEXT   *DispatchContext
  );

////////////////////////////////////////////
```

```
EFI_DRIVER_ENTRY_POINT(InitializeChild)

EFI_STATUS
InitializeChild (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
/*++

Routine Description:

  Initializes the SMM Handler Driver

Arguments:

  ImageHandle -

  SystemTable -

Returns:

  None

--*/
{
  EFI_STATUS                         Status;
  BOOLEAN                            InSmm;
  EFI_HANDLE                         Handle;
  UINT8*                             Buffer;
  UINTN                              BufferSize;

  EFI_SMM_SW_DISPATCH_CONTEXT        SwContext = { 0 };
  EFI_HANDLE                         SwHandle = 0;

  Status = BufferSize = InSmm = 0;
  Handle = Buffer = NULL;

  //
  // Initialize the EFI Runtime Library
  //
  EfiInitializeSmmDriverLib (ImageHandle, SystemTable);

  Status = gBS->LocateProtocol(&gEfiSmmBaseProtocolGuid, NULL, &mSmmBase);
  if (EFI_ERROR(Status)) {
    return Status;
  }

  mSmmBase->InSmm(mSmmBase, &InSmm);

  if (!InSmm) {
    //
    // This driver is dispatched by DXE, so first call to this
    // driver will not be in SMM.  We need to load this driver
    // into SMRAM and then generate an SMI to initialize data
    // structures in SMRAM.
    //

    // Load this driver's image to memory
    Status = GetFvImage(&mChildFileGuid, &Buffer, &BufferSize);
    if (!EFI_ERROR(Status)) {
      // Load the image in memory to SMRAM; it will automatically
      // generate the SMI.
      mSmmBase->Register(mSmmBase, NULL, Buffer, BufferSize, &Handle,
FALSE);
      gBS->FreePool(Buffer);
    }

  } else {
    //
    // Great!  We're now in SMM!
    //
```

```
    // Initialize global variables
    mSmmBase->GetSmstLocation(mSmmBase, &mSmst);

    // Locate SwDispatch protocol
    Status = gBS->LocateProtocol(&gEfiSmmSwDispatchProtocolGuid, NULL,
&mSwDispatch);
    if (EFI_ERROR(Status)) {
      DEBUG(( EFI_D_ERROR, "Couldn't find SmmSwDispatch protocol: %r\n",
Status));
      return Status;
    }

    //
    // Register for callbacks
    //

    // Pick a value for the context and register for it
    SwContext.SwSmiInputValue = your value written to software SMI port;
    Status = mSwDispatch->Register( mSwDispatch, SwCallback, &SwContext,
&SwHandle );
    ASSERT_EFI_ERROR( Status );


    // If your SMM handler can be entered via multiple soft SMM values
    // then repeat the above 3 lines per additional value,


  }

  return EFI_SUCCESS;
}

/////////////////////////////////////////////
// Callback functions

VOID
SwCallback (
  IN  EFI_HANDLE                    DispatchHandle,
  IN  EFI_SMM_SW_DISPATCH_CONTEXT   *DispatchContext
  )
{
  DEBUG(( EFI_D_ERROR, "                 Sw SMI captured w/ context
0x%02x\n", DispatchContext->SwSmiInputValue));
  // place your SMM code here

}

/////////////////////////////////////////////
```

4.2 Example of a Dummy EFI Hardware SMM Child Driver

User defined areas are highlighted like this.

```
/*++


Module Name:

    YourName.c

Abstract:

    This is a generic template for a child of the IchSmm driver.

Revision History

--*/

#include "Efi.h"
#include "EfiRuntimeLib.h"

#include "GetFvImage.h"

#include EFI_PROTOCOL_CONSUMER(SmmBase)
#include EFI_PROTOCOL_CONSUMER(FirmwareVolume)
#include EFI_PROTOCOL_CONSUMER(YourFileDispatch)


// GUID for the FV file that this source file gets compiled into
EFI_GUID  mChildFileGuid = { Your GUID here };

//////////////////////////////////////////////
// Callback function prototypes

VOID
YourCallback (
  IN  EFI_HANDLE                    DispatchHandle,
  IN  EFI_SMM_YOUR_DISPATCH_CONTEXT  *DispatchContext
  );
```

```
///////////////////////////////////////////

EFI_DRIVER_ENTRY_POINT(InitializeChild)

EFI_STATUS
InitializeChild (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
/*++

Routine Description:

  Initializes the SMM Handler Driver

Arguments:

  ImageHandle -

  SystemTable -

Returns:

  None

--*/
{
  Your code here
  //
  // Initialize the EFI Runtime Library
  //
  EfiInitializeSmmDriverLib (ImageHandle, SystemTable);

  Status = gBS->LocateProtocol(&gEfiSmmBaseProtocolGuid, NULL, &mSmmBase);
  if (EFI_ERROR(Status)) {
    return Status;
  }

  mSmmBase->InSmm(mSmmBase, &InSmm);

  if (!InSmm) {
    //
    // This driver is dispatched by DXE, so first call to this driver
    // will not be in SMM.  We need to load this driver into SMRAM and
    // then generate an SMI to initialize data structures in SMRAM.
    //

    // Load this driver's image to memory
    Status = GetFvImage(&mChildFileGuid, &Buffer, &BufferSize);
    if (!EFI_ERROR(Status)) {
      // Load the image in memory to SMRAM; it will automatically
      // generate the SMI.
      mSmmBase->Register(mSmmBase, NULL, Buffer, BufferSize, &Handle,
FALSE);
      gBS->FreePool(Buffer);
    }

  } else {
    //
    // Great!  We're now in SMM!
    //

    // Initialize global variables
    mSmmBase->GetSmstLocation(mSmmBase, &mSmst);

    // Get Your  protocol
    Status = gBS->LocateProtocol(&gEfiSmmYourGuid, NULL, &mYourFile);
    if (EFI_ERROR(Status)) {
      DEBUG(( EFI_D_ERROR, "Couldn't find Your File protocol: %r\n",
Status));
      return Status;
    }
```

```
    // Register for the Your event. This defines the hardware path
    // and bits associated with the hardware SMM. These are defined
    // by the Framework.
    //
    YourContext.Type = Framework assigned type;
    YourContext.Device = (EFI_DEVICE_PATH_PROTOCOL*)&Framework assigned
path;

    Status = mYourFile->Register( mYourFile, YourCallback, &YourContext,
&YourHandle );
    if (EFI_ERROR(Status)) {
      DEBUG(( EFI_D_ERROR, "Couldn't register for callback: %r\n",
Status));
      return Status;
    }

    DEBUG(( EFI_D_ERROR, "Your file device path address: 0x%x\n",
      &YourPATH
      ));

  }

  return EFI_SUCCESS;
}

////////////////////////////////////////////
// Callback functions

VOID
YourCallback (
  IN  EFI_HANDLE                    DispatchHandle,
  IN  EFI_SMM_USB_DISPATCH_CONTEXT  *DispatchContext
  )
{

  DEBUG(( EFI_D_ERROR, "                 Your SMI captured T%d D%x\n",
    DispatchContext->Type,
    DispatchContext->Device
    ));

Your code here}

////////////////////////////////////////////
```

5
Legacy BIOS References

5.1 BIOS INTs

This document lists only the INTs to be supported and does not list all subfunctions unless they are not required. Refer to the *IBM* Personal System/2 and Personal Computer BIOS Interface Technical Reference* or any of the AMI* or Phoenix* BIOS manuals for full information on all subfunctions.

INT 0x02 - NMI

There needs to be a NMI handler.

INT 0x05 - Print Screen

This INT must be supported. Note that this INT modifies memory location 50:00.

Memory Location 50:00

This is a Byte memory location. A value of 0x00 indicates that the print screen successfully completed or was not invoked. A value of 0x01 indicates that a print screen is in progress and subsequent print screens are ignored. A value of 0xFF indicates that the print screen terminated due to an error.

INT 0x08 - System Timer

This INT must be supported. Note that this INT modifies memory locations 40:6C, 40:70, 40:40, and 40:3F. It also invokes software INT 1C.

Memory Location 40:6C

This location is a Dword memory location. The value is incremented every INT 08 tick or 18.2 times a second. The memory location is reset to 0x00000000 when a 24-hour duration has elapsed.

Memory Location 40:70

This location is a Byte memory location. This location has a value of 0x00 until a 24-hour duration has elapsed. It is then set to 0x01. The byte must be manually reset back to 0x00.

Memory Location 40:40

This location is a Byte memory location. The value is decremented every INT 08 tick or 18.2 times a second. If the timer goes to 0x00, the floppy motor is turned off and resets the floppy flags in memory location 40:3F.

Memory Location 40:3F

This location is a Byte memory location. Bit 1 is set if drive B motor is on. Bit 0 is set if drive A motor is on.

INT 0x09 - Keyboard

This INT must be supported. It is called on every make or break keystroke. The 32-byte buffer starting at 40:1E is updated at the address pointed by the keyboard-buffer tail pointer. The keyboard-buffer tail pointer at memory location 40:1C is incremented by 2 unless it extends past the keyboard-buffer, in which case it wraps. When a key is read, the keyboard-buffer head pointer at memory location 40:1A is incremented by 2 unless it extends pass the keyboard-buffer, in which case it wraps. Special keys such as CTRL, ALT, or Shift update the status at memory location 40:17, 40:18 and 40:96. A CTRL-ALT-DELETE key sequence sets the reset flag at memory location 40:72 to 0x1234 and jumps to the reset vector.

Pressing the Pause key causes the interrupt handler to loop until a valid ASCII keystroke occurs.

Pressing the Print Screen key causes an INT 0x05 to be issued.

A CTRL-BREAK sequence causes INT 0x1B to be issued.

Pressing the SysReq key causes INT 0x15 Function 0x85 (System Request Key Pressed) to be issued.

Any make keystroke causes INT 0x15 Function 0x91, Subfunction 0x02 (Interrupt complete from Keyboard) to be issued.

After any scan code is read from I/O port 0x60 and INT 0x15, Function 0x4F (Keyboard Intercept) is issued. An EOI is issued upon returning from the Keyboard Intercept.

Memory Location 40:1E

This location is the start of a 32-byte keyboard buffer.

Memory Location 40:1A

This location is a Word memory location. It points to the next character in the keyboard buffer.

Memory Location 40:1C

This location is a Word memory location. It points to the last character in the keyboard buffer. If the value equals the value in memory location 40:1A, the keyboard buffer is empty. If the value is two bytes from the contents of memory location 40:1A, the keyboard buffer is full.

Memory Location 40:17

This location is a Byte memory location and contains the keyboard status byte.

Memory Location 40:18

This location is a Byte memory location and contains the extended keyboard status byte.

Memory Location 40:96

This location is a Word memory location and contains the extended keyboard status.

Memory Location 40:72

This location is a Word memory location and contains the soft reset flag.

INT 0x10 - Video

This INT is supported by the video OpROM. There is no native planar BIOS support.

INT 0x11 - Equipment Determination

This INT must be supported. This INT returns the data at memory location 40:10.

Memory Location 40:10

This location is a Word memory location and contains the equipment list.

INT 0x12 - Base Memory Size

This INT must be supported and returns the value at memory location 40:13.

Memory Location 40:13

This location is a Word memory location and contains the amount of memory up to 640 KB. It is set to 0x280 regardless of the Extended BIOS Data Area (EBDA) size because the 512 KB option has been obsoleted.

INT 0x13 - HDD and Floppy Diskette Services

This INT must be supported, including the Microsoft* extensions. IDE, floppy diskette, ATAPI, ATA, El Torito, and ARMD drives must be supported. Note that if non-floppy controllers are present, INT 0x40 must be supported.

INT 0x14 - Serial Communication Services

This INT must be supported.

INT 0x15 - System Services

This INT must be supported. Subfunctions are those defined by the *IBM* Personal System/2 and Personal Computer BIOS Interface Technical Reference* manual. Additional subfunction support is OEM and/or IBV dependent.

INT 0x16 - Keyboard Services

This INT must be supported.

INT 0x17 - Printer Services

This INT must be supported.

INT 0x1A - System-Timer Services

This INT must be supported, including the PCI BIOS extensions.

INT 0x1B - CTRL- BREAK Services

The BIOS sets this to IRET and the OS hooks it.

INT 0x1C - Periodic Timer Interrupt

The BIOS sets this to IRET and the OS hooks it.

INT 0x1D - Video Parameter Table

Set by the video BIOS.

INT 0x1E - Floppy Diskette Drive Parameters

Points to an 11-byte data structure.

INT 0x1F - Video Graphics Characters

Set by the video BIOS.

INT 0x40 - Floppy Diskette Services

This INT must be supported if non-floppy controllers are present.

INT 0x41 - HDD C: Drive Parameters

Points to a 16-byte data structure for drive C:.

INT 0x46 - HDD D: Drive Parameters

Points to a 16-byte data structure for drive D:.

5.2 Fixed BIOS Entry Points

The fixed entry points in F000: xxxx must be supported for traditional reasons. The table below lists the fixed BIOS entry points.

Table 14 Fixed BIOS Entry Points

Location	Description
F000: E05B	POST Entry Point
F000: E2C3	NMI Entry Point
F000: E401	HDD Parameter Table
F000: E6F2	INT 19 Entry Point
F000: E6F5	Configuration Data Table
F000: E729	Baud Rate Generator Table
F000: E739	INT 14 Entry Point
F000: E82E	INT 16 Entry Point
F000: E987	INT 09 Entry Point
F000: EC59	INT 13 Floppy Entry Point
F000: EF57	INT 0E Entry Point
F000: EFC7	Floppy Disk Controller Parameter Table
F000: EFD	INT 17
F000: F065	INT Video
F000: F0A4	MDA and CGA Video Parameter Table INT 1D
F000: F841	INT 12 Entry Point
F000: F84D	INT 11 Entry Point
F000: F859	INT 15 Entry Point
F000: FA6E	Low 128 character of graphic video font
F000: FE6E	INT 1A Entry Point
F000: FEA	INT 08 Entry Point
F000: FF53	Dummy Interrupt Handler
F000: FF54	INT 05 Print Screen Entry Point
F000: FFF0	Power-On Entry Point
F000: FFF5	ROM Date in ASCII “MM/DD/YY” for 8 characters
F000: FFFE	System Model 0xFC

5.3 Fixed CMOS Locations

The table below lists the fixed CMOS locations.

Table 15 Fixed CMOS Locations

Start Location	Length in bytes	Description	Modified by Traditional BIOS	Comments
0x00	10	RTC	Yes	INT1A
0x0A	6	CMOS Status	No	
0x10	1	Floppy drive type	No	
0x12	1	Hard Disk	No	
0x14	1	Equipment byte	No	
0x15	2	Base memory size	No	
0x17	2	Extended memory size	No	
0x19		Hard disk C drive type	No	
0x1A		Hard Disk D drive type	No	
0x2E	2	Standard CMOS checksum	Yes	If any location 0x10 through 0x2D is changed
0x30	2	Extended memory found by BIOS	No	
0x32	1	Century byte	Yes	On Roll over, IN 1A
0x33	1	Information Flag	No	Bit 0 =1 - Cache good. Not check summed.
0x3E	2	Extended CMOS checksum	Yes	If any location 0x30 through 0x7F is changed and in check summed region.

Notes:

If CMOS is not supported by standard EFI, then the bytes that are labeled as not modified by the traditional BIOS must be initialized by EfiCompatibility but are not modified at runtime.

The CSM may use other CMOS bytes. If standard EFI supports CMOS, then the CMOS usage most not conflict.

CMOS locations greater than 0x33 are up to the implementer as far as inclusion/exclusion in a checksum range.

5.4 BDA and EBDA Memory Addresses

The BIOS Data Area (BDA) starts at 40:0 and is 257 bytes in length. Byte 40:100 is byted by INT 0x05.

Table 16 BIOS Data Area

Start Location	Length in bytes	Description	Modified by Legacy BIOS	INT Using It	Comments
0x00	2	COM 1 base address	No	14	
0x02	2	COM 2 base address	No	14	
0x04	2	COM 3 base address	No	14	
0x06	2	COM 4 base address	No	14	
0x08	2	LPT 1 base address	No	17	
0x0A	2	LPT 2 base address	No	17	
0x0C	2	LPT 3 base address	No	17	
0x0E	2	EBDA segment	No		
0x10	2	Installed hardware	No	11	
0x12	1	Reserved	No		
0x13	2	Base memory size	No	12	
0x15	2	Reserved	No		
0x17	1	Keyboard control 1	Yes	16	
0x18	1	Keyboard control 2	Yes	16	
0x19	1	Work area for ALT key	Yes	16	
0x1A	2	Keyboard-buffer Head	Yes	16	
0x1C	2	Keyboard-buffer Tail	Yes	16	
0x1E	32	Keyboard Buffer	Yes	16	
0x3E	1	Floppy recalibrate status	Yes	13	
0x3F	1	Floppy motor status	Yes	13	
0x40	1	Floppy motor timeout	Yes	13	
0x41	1	Floppy operation status	Yes	13	
0x42	7	Floppy controller status	Yes	13	
0x49	30	Video info	No	10	
0x67	4	POST re-entry ptr			
0x6B	1	Last Unexpected interrupt	Yes		
0x6C	4	Timer Counter	Yes	1A	
0x70	1	Timer Overflow	Yes	1A	
0x71	1	Break key state	Yes	16	
0x72	2	Reset Flag	Yes		
0x74	1	HDD operation status	Yes	13	

Start Location	Length in bytes	Description	Modified by Legacy BIOS	INT Using It	Comments
0x75	1	Number of HDDs attached	No	13	
0x76	2	Reserved	No	13	
0x78	1	LPT 1 time-out	Yes	14	
0x79	1	LPT 2 time-out	Yes	14	
0x7A	1	LPT 3 time-out	Yes	14	
0x7B	1	Reserved	No		
0x7C	1	COM 1 time-out	Yes		
0x7D	1	COM 2 time-out	Yes		
0x7E	1	COM 3 time-out	Yes		
0x7F	1	COM 4 time-out	Yes		
0x80	2	Keyboard buffer start ptr		16	
0x82	2	Keyboard buffer end ptr		16	
0x84	7	Video info	No	10	
0x89	2	Reserved	No		
0x8B	1	Floppy media control	Yes	13	
0x8C	1	HDD Controller status	Yes	13	
0x8D	1	HDD Controller error status	Yes	13	
0x8E	1	HDD Interrupt control	Yes	13	
0x8F	1	Reserved	No		
0x90	1	Floppy 0 media status	Yes	13	
0x91	1	Floppy 1 media status	Yes	13	
0x92	1	Floppy 2 media status	No?	13	
0x93	1	Floppy 3 media status	No?	13	
0x94	1	Drive 0 current cylinder	Yes	13	
0x95	1	Drive 1 current cylinder	Yes	13	
0x96	1	Keyboard mode state & flags	Yes	16	
0x97	1	Keyboard LED flags	Yes	16	
0x98	2	User Wait flag offset	Yes	15	
0x9A	2	User wait flag segment	Yes	15	
0x9C	2	Low word of user wait count	Yes	15	
0x9E	2	High word of user wait count	Yes	15	
0xA0	1	Wait active flag	Yes	15	
0xA1	7	Reserved	No		
0xA8	4	Video info	No	10	

Start Location	Length in bytes	Description	Modified by Legacy BIOS	INT Using It	Comments
0xAC	0x54	Reserved	No		
0x100	1	Print Screen status	Yes	05	

5.5 EBDA (Extended BIOS Data Area)

This area starts at the segment pointed to by the contents of 40:0E.

Table 17 Extended BIOS Data Area

Start Location	Length in bytes	Description	Modified by Legacy BIOS	Comments
0x00	1	Length of EBDA in KB	No	
0x01	32	Reserved	No	
0x17	1	Number of POST errors	No	
0x18	5	POST error log	No	
0x22	4	Mouse Driver Ptr	No	INT74; Compatibility16 calls this pointer
0x26	1	Mouse flag byte 1	Yes	INT74
0x27	1	Mouse flag byte 2	Yes	INT74
0x28	8	Mouse data	Yes	INT74
0x30	0x3D0	Reserved	No	

5.6 IA-32 and Itanium Processor Family Interrupts

5.6.1 EFI Environment

An EFI-only environment normally only has the timer interrupt hooked. The processor traps, exceptions and faults are also trapped. There is only one supported hardware interrupt for IA-32 (Timer interrupt). For the Itanium® processor family, the only supported hardware interrupt is a processor counter ITC generated interrupt. There are no software interrupts supported by either processor family.

5.6.2 IA-32

Traditionally IRQ0 through IRQ7 are allocated to INT 0x08 through INT 0x0F, and IRQ8 through IRQ15 are allocated to INT 0x70 through INT 0x77. The traditional allocation of INT 0x08 through INT 0x0F overlay with processor faults, exceptions and traps. It is safe to move IRQ0 through IRQ7 to INT 0x68 through 0x6F, thus leaving INT 0x08 through INT 0x0F free for the processor faults, exceptions and traps. The only interrupt unmasked in the PIC registers 0x21 and 0xA1 should be the timer or IRQ0. APICs in non-8259 mode are platform specific and outside the scope of this document.

Note: *SYNC1 IRQ0–7 are at traditional INTs and need to be moved.*

5.6.2.1 IA-32 Faults, Exceptions, and Traps

The table below lists the IA-32 faults, exceptions, and traps.

Table 18 IA-32 Faults, Exceptions, and Traps

INT	Fault, Exception, or Trap
00	Divide by Zero Fault
01	Code Breakpoint Fault
02	NMI Trap
03	INT 3 Breakpoint Trap
04	Overflow Exception
05	Bounds Fault
06	Invalid Opcode Fault
07	No Math Coprocessor or Device Not Available Fault
08	Double Fault
09	Coprocessor Segment Overrun – Obsolete
0A	Invalid TSS Fault
0B	Segment Not Present Fault
0C	Stack Segment Fault
0D	General Protection Fault
0E	Page Fault
0F	Reserved
10	Floating-Point Error
11	Alignment Check Fault
12	Machine Check
13-1F	Reserved by Intel

5.6.2.2 IA-32 Interrupts

The table below lists the IA-32 interrupts.

Table 19 IA-32 Interrupts

INT	Description
20-67	Unused
68	IRQ 0 – Timer interrupt
69	IRQ 1 - Unused
6A	IRQ 2 - Unused
6B	IRQ 3 - Unused
6C	IRQ 4 - Unused
6D	IRQ 5 - Unused
6E	IRQ 6 - Unused
6F	IRQ 7 - Unused
70	IRQ 8 - Unused
71	IRQ 9 - Unused
72	IRQ 10 - Unused
73	IRQ 11 - Unused
74	IRQ 12 - Unused
75	IRQ 13 - Unused
76	IRQ 14 - Unused
77	IRQ 15 - Unused
78-FF	Unused

5.6.3 Intel® Itanium® Processor Family

The Itanium® processor family has two generic types of interrupts:

- PAL-based interrupts
- Interruption Vector Table (IVA)–based interrupts

PAL-based interrupts are handled by the PAL firmware, system firmware, or possibly the OS. IVA-based interrupts are handled by the system firmware and operating system. The following topics discuss these interrupts in more detail.

5.6.3.1 PAL-Based Interrupts

The table below lists the PAL-based interrupts for the Itanium® processor family.

Table 20 PAL-Based Interrupts

Type	Name	PALE Entry	Description
Abort	Machine Checks (MCA)	PALE_CHECK	An immediate action hardware error has occurred.
Abort	Processor Reset	PALE_RESET	A processor has been powered on or a reset request sent to it.
Initialization interrupts	INIT	PALE_INIT	A processor has received an initialization interrupt.
Platform Management interrupts	PMI	PALE_PMI	A platform management request has been received..

5.6.3.2 IVA-Based Interrupts

Itanium processors support a SAPIC component and an internal ITC (Interval Timer Counter – AR44), which counts up at a fixed relationship to the processor clock frequency. The controlling parameter for this internally delivered interrupt can be programmed into ITV (CR72). When the ITC count reaches the value programmed into the Interval Timer Match Register (ITM-CR1), the interval timer interrupt is raised. In the SAPIC mode, they are directly delivered internally to the processor. This mechanism is used to get the timer tick interrupt that is needed for EFI core operation.

Interruption Vector Table (IVA)–based interrupts function very differently in the Itanium processor family but allow the management of traditional 8259-based interrupts. When a hardware interrupt occurs, the processor switches to an alternate bank of registers, loads the preinterrupt context to several control registers (such as ipsr, iip, and so on), and then branches to a location pointed by cr.IVA + 0x3000. At this location, the hardware interrupt management code starts executing. This code will read cr.ivr and if the vector is 00, then it is a traditional 8259-generated interrupt. If it is nonzero, then it is a SAPIC-programmed interrupt. The ITC interrupt mentioned earlier is one such thing with a distinct SAPIC-supplied nonzero vector.

If it is a traditional interrupt, then the Itanium® architecture interrupt handler code will do a non-cached one-byte load from a special cycle location at offset 0x1e00 from the base of processor interrupt block region, which has been programmed into the processor through a PAL call. Either the internal bus unit or the chipset would then recognize the special cycle (for the Itanium processor family, the logic is in the processor) and will produce two INTA bus cycles to 8259. The first cycle is ignored by 8259 as it is programmed to 8086 mode by the CSM code/8259 INIT code (the first cycle will produce a call 8085 opcode if 8259 is programmed into 8085 mode, which is not the case). The 8259 will respond to the second INTA cycle and will send the vector up the bus, and the Itanium architecture code will read it by its special cycle one-byte load.

This Itanium architecture code has an option of processing this vector. The CSM design must be such that the code reflects this option to 16-bit IA-32 code. This vector number will be multiplied by 4 and code segment and offset shall be read. Itanium architecture will save the machine context, including floating point registers, and then loads the CS and IP value to the appropriate Itanium processor family registers and prepares the 16-bit IA-32 code environment. The new stack and then the Itanium architecture code is provided. Then it will branch to the Itanium architecture code with a special br.ia instruction. The saving of the context is necessary as IVE microcode uses all the Itanium architecture registers.

When the IA-32 handler does an Iret instruction, this instruction is trapped by Itanium architecture and the Itanium architecture trap handler restores the caller context and returns through an RFI.

It is possible that IA-32 code may not do an Iret but does “ret 2.” There are several ways to handle this situation. One way is to let the Itanium architecture handler point the IA-32 stack to a deliberately and intentionally faulting instruction such as rep:HLT (IA-32 opcode 0xf, 0xf4) and then take control in the Itanium architecture fault handler to restore the context.

5.6.3.3 Assumptions

The previous topics in this section assumed that the platform has an 8259 PIC and an IVE (Intel® value-added engine core that executes most of the IA-32 instructions). But future processors may not have those elements. In that case, traditional mode can be handled only by an IA-32 instruction emulator. One way of doing this handling is to set the psr (processor status register) in such a way that execution of all IA-32 instructions fault into native code and hence get emulated.

IVA-based interrupts include external interrupts, NMI, faults and traps. A unique vector number 0,2,0x10 through 0xFF defines external interrupts. The list below in the Description field is a list of IVA-based interrupts that may be used by the Framework.

Table 21 IVA-Based Interrupts Useable in the Framework

Type	Name	Description
External Interrupts	INT 0	Unused
External Interrupt	INT 2	Unused
External Interrupt	INT 0x10- 0xFF	Unused
	Alternate Data TLB	
	Alternate Instruction TLB	
	Break Instruction	Used
	Data Access Rights	
	Data Access-Bit	
	Data Key Miss	
	Data Nested TLB	
	Data TLB	
	Debug	Used
	Dirty-Bit	
	Disable FP-Register	
	Floating-point Fault	
	Floating-point trap	
	General Exception	Used
	IA-32 Exception	General IA-32 fault
	IA-32 Interrupt	IA-32 invalid opcode
	IA-32 Interrupt	IA-32 software interrupt

Type	Name	Description
	Instruction Access Rights	
	Instruction Access-Bit	
	Instruction Key Miss	
	Instruction TLB	
	Key Permission	
	Lower-Privilege Transfer Trap	
	NaT Consumption	
	Page Not Present	
	Single Step Trap	Used
	Speculation	
	Taken Branch Trap	
	Unaligned Reference	
	Unsupported Data Reference	
	VHPT Translation	

5.6.4 Mixed EFI and Traditional Environment

The mixed EFI and traditional environment imposes several complexities over the EFI-only environment. The main complexity is a transition to/from 32-bit/16-bit mode. An additional complexity is that the traditional environment must handle many more interrupts as devices are interrupt driven versus polled. The hardware and software interrupts that are used depend upon the traditional OpROMs invoked. The following is a high-level view and the reader should refer to the appropriate AMI* or Phoenix* BIOS specifications for details.

1. Set the appropriate flags as current operating mode.
2. Allocate area on the 16-bit stack for registers.
3. Copy current EFI registers to 16-bit stack.
4. Save the current interrupt state.
5. Disable interrupts.
6. Change the interrupt mask to traditional mode.
7. Invoke EFI to legacy thunk code.
8. Perform legacy operations.
9. Disable interrupts; traditional code may re-enable them.
10. Change the interrupt mask to EFI mode.
11. Thunk back to EFI.
12. Restore the interrupt state.

13. Invoke the timer tick interrupt, if needed.

14. Copy the 16-bit register stack to the EFI stack.

15. Return the carry flag state for successful/unsuccessful completion.

5.6.4.1 IA-32

The Legacy section supports all the traditional BIOS software and hardware interrupts.

5.6.4.2 Itanium Processor Family

See the comments on executing 16-bit code. There is some added complexity because, at a minimum, 8 nested interrupts need to be handled, and a few instructions such as lGDT, CLI, and STI, need to be emulated. See the SAL specification for a complete list. Also, we have to care for software interrupts, which can be done by careful thunking. Also, special software interrupts such as AH=88, INT15, and other interrupts that need to go to protected IA-32 mode must be serviced in 64-bit native mode itself by filtering them out.

5.6.4.2.1 E820 CALL BUILDING

The E820 call runs in pure 16-bit real mode, so none of the EFI memory records or SAL system records can be touched as they exist above 1 MB. So, we need to steal some EBDA and copy them into it and modify the E820 call to translate these records from EBDA into E820 records.

5.6.5 Traditional-Only Environment

The traditional-only environment is similar to the mixed EFI and traditional operation, except that once the EFI code thunks into traditional mode, the traditional code never returns to EFI mode. The traditional hardware and software interrupts are supported.

Note the following information for the Itanium® processor family in the traditional-only environment:

- There must be a PAL emulation layer that is invoked by the PAL_enter_IA32 call.
- If the traditional OS returns that it is unable to load or that the OS was not found, the code must go back to the EFI loader. To go back to native 64-bit code, execute a special instruction called “jmpe” in the traditional 32-bit mode. This instruction will abort the PAL emulation layer and will return to the PAL emulation retiring address that is registered by native code before invoking the PAL emulation layer. The native code will take over from there.

6 Glossary

16-bit legacy: The traditional PC environment and includes traditional OpROMs and Compatibility16 code.

BDA: BIOS Data Area.

Compatibility16: The traditional BIOS with POST and BIOS Setup removed. Executes in 16-bit real mode.

CompatibilitySmm: Any IBV-provided SMM code to perform traditional functions that are not provided by EFI.

CSM: Compatibility Support Module. The combination of EfiCompatibility, CompatibilitySmm, and Compatibility16.

EBDA: Extended BIOS Data Area.

EfiCompatibility: 32-bit EFI code to generate data for traditional BIOS interfaces or EFI Compatibility Support Module (CSM) drivers or code to invoke traditional BIOS services.

EOI: End of Interrupt.

IDT: Interrupt Descriptor Table.

ITC: Interval Timer Counter.

ITM: Interval Timer Match.

IVA: Interruption Vector Table.

NV: Nonvolatile.

PIC: (2) Programmable Interrupt Controller.

PMM: Post Memory Manager.

RACBA: Reserve Area Boot Code Address.

traditional OpROM: 16-bit OpROMs that are executed in real mode.