intel®

Extensible Firmware Interface Specification

Version 1.10

December 1, 2002

Acknowledgements

The UGA Protocol sections of this specification were developed in close consultation with Microsoft as part of the Universal Graphics Adapter (UGA) initiative. Microsoft has made significant contributions to the interface definitions presented here to ensure that they will work well with video adapters supporting the Microsoft UGA specification. These efforts are gratefully acknowledged.

The EFI Byte Code Virtual Machine sections (Chapter 19) of this specification were developed in close consultation with Microsoft, LSI, Hewlett Packard, Compaq, and Phoenix Technologies. The efforts of all contributors to these sections are gratefully acknowledged.

Intel order number: A79614-002

History

Revision	Revision History	Date
1.0	Official release of EFI 1.10 Specification, revision 1.0.	12/01/02

Contents

14 Protocols — USB Support

References

Figures

Tables

1 Introduction

This *Extensible Firmware Interface* (hereafter known as EFI) *Specification* describes an interface between the operating system (OS) and the platform firmware. The interface is in the form of data tables that contain platform-related information, and boot and runtime service calls that are available to the OS loader and the OS. Together, these provide a standard environment for booting an OS.

The EFI specification is designed as a pure interface specification. As such, the specification defines the set of interfaces and structures that platform firmware must implement. Similarly, the specification defines the set of interfaces and structures that the OS may use in booting. How either the firmware developer chooses to implement the required elements or the OS developer chooses to make use of those interfaces and structures is an implementation decision left for the developer.

The intent of this specification is to define a way for the OS and platform firmware to communicate only information necessary to support the OS boot process. This is accomplished through a formal and complete abstract specification of the software-visible interface presented to the OS by the platform and firmware.

Using this formal definition, a shrink-wrap OS intended to run on Intel® architecture-based platforms will be able to boot on a variety of system designs without further platform or OS customization. The definition will also allow for platform innovation to introduce new features and functionality that enhance platform capability without requiring new code to be written in the OS boot sequence.

Furthermore, an abstract specification opens a route to replace legacy devices and firmware code over time. New device types and associated code can provide equivalent functionality through the same defined abstract interface, again without impact on the OS boot support code.

The EFI specification is primarily intended for the next generation of 32-bit Intel architecture (IA-32) and Itanium®-based computers. Thus, the specification is applicable to a full range of hardware platforms from mobile systems to servers. The specification provides a core set of services along with a selection of protocol interfaces. The selection of protocol interfaces can evolve over time to be optimized for various platform market segments. At the same time the specification allows maximum extensibility and customization abilities for OEMs to allow differentiation. In this, the purpose of EFI is to define an evolutionary path from the traditional "PC-AT"-style boot world into a legacy-API free environment.

1.1 EFI Driver Model Extensions

Access to boot devices is provided through a set of protocol interfaces. The *EFI 1.02 Specification* describes these protocol interfaces in detail. However, it does not describe how these protocol interfaces are produced by the system firmware. The *EFI 1.10 Specification* includes extensions to the *EFI 1.02 Specification* that describe the *EFI Driver Model* along with additional protocol interfaces that provide access to a richer set of boot devices. One purpose of the *EFI Driver Model* is to provide a replacement for "PC-AT"-style option ROMs. It is important to point out that drivers written to the *EFI Driver Model* are designed to access boot devices in the preboot environment. They are not designed to replace the high performance OS specific drivers. The *EFI 1.10 Specification* is designed to be backward compatible with the *EFI 1.02 Specification*. This means that any EFI applications or drivers written to the *EFI 1.02 Specification* will continue to function on system firmware that complies with the *EFI 1.10 Specification*.

The *EFI Driver Model* is designed to support the execution of modular pieces of code, also known as drivers that run in the preboot environment. These drivers may manage or control hardware buses and devices on the platform or they may provide some software derived platform specific service.

The *EFI Driver Model* is designed to extend the *EFI Specification* in a way that supports device drivers and bus drivers. These extensions are provided in the form of new protocols, new boot services, and updated EFI boot services that are backward compatible with their original versions. The *EFI Driver Model* also contains information required by EFI driver writers to design and implement any combination of bus drivers and device drivers that a platform may need to boot an EFI compliant OS.

The *EFI Driver Model* is designed to be generic and can be adapted to any type of bus or device. The *EFI 1.10 Specification* describes how to write PCI bus drivers, PCI device drivers, USB bus drivers, USB device drivers, and SCSI drivers. Additions details are provided that allow EFI drivers to be stored in PCI option ROMs while maintaining compatibility with legacy option ROM images.

One of the design goals in the *EFI 1.10 Specification* is to keep the driver images as small as possible. However, if a driver is required to support multiple processor architectures, a driver object file would have to be shipped for each supported processor architecture. To address this space issue, the *EFI 1.10 Specification* also defines the *EFI Byte Code Virtual Machine*. An EFI driver can be compiled into a single EFI Byte Code Virtual Machine object file. EFI 1.10 complaint firmware must contain an EFI Byte Code interpreter. This allows a single EFI Byte Code object file to be shipped that supports multiple processor architectures. Another space saving technique is the use of compression. The *EFI 1.10 Specification* defines compression and decompression algorithms that may be used to reduce the size of EFI drivers, and thus reduce the overhead when EFI drivers are stored in ROM devices.

The information contained in the *EFI 1.10 Specification* can be used by OSVs, IHVs, OEMs, and firmware vendors to design and implement EFI firmware, EFI drivers that produce standard protocol interfaces, and EFI operating system loaders that can be used to boot EFI compliant operating systems.

1.2 Overview

This EFI 1.10 Specification is organized as listed in Table 1-1.

Table 1-1. Organization of the EFI Specification

Chapter/Appendix	Description
1. Introduction	Introduces the EFI Specification and topics related to using the specification.
2. Overview	Describes the major components of EFI, including the boot manager, firmware core, calling conventions, protocols, and requirements.
3. Boot Manager	Describes the boot manager, which is used to load EFI drivers and EFI applications.
4. EFI System Table	Describes the EFI System Table that is passed to every EFI driver and EFI application.
5. Services — Boot Services	Contains the definitions of the fundamental services that are present in an EFI-compliant system before an OS is booted.
6. Services — Runtime Services	Contains definitions for the fundamental services that are present in an EFI-compliant system before and after an OS is booted.
7. Protocols — EFI Loaded Image	Defines the EFI Loaded Image Protocol that describes an EFI Image that has been loaded into memory.
8. Protocols — Device Path Protocol	Defines the device path protocol and provides the information needed to construct and manage device paths in the EFI environment.
9. Protocols — EFI Driver Model	Describes a generic driver model for EFI. This includes the set of services and protocols that apply to every bus and device type. These protocols include the Driver Binding Protocol, the Platform Driver Override Protocol, the Bus Specific Driver Override Protocol, the Driver Diagnostics Protocol, the Driver Configuration Protocol, and the Component Name Protocol.
10. Protocols — Console Support	Defines the Console I/O protocols, which handle input and output of text-based information intended for the system user while executing in the EFI boot services environment. These protocols include the Simple Input Protocol, the Simple Text Output Protocol, the Universal Graphics Adapter (UGA) Protocol, the Simple Pointer Protocol, and the Serial I/O Protocol.

continued

Table 1-1. Organization of the EFI Specification (continued)

Chapter/Appendix	Description
11. Protocols — Bootable Image Support	Defines the protocols that provide access to bootable images while executing in the EFI boot services environment. It also describes the supported disk layouts including MBR, El Torito, and the Guided Partition Table (GPT). These protocols include the Load File Protocol, the Simple File System Protocol, the Disk I/O Protocol, the Block I/O Protocol, and the Unicode Collation Protocol.
12. Protocols — PCI Bus Support	Defines PCI Bus Drivers, PCI Device Drivers, and PCI Option ROM layouts. The protocols described include the PCI Root Bridge I/O Protocol and the PCI I/O Protocol.
13. Protocols — SCSI Bus Support	Defines the SCSI Pass Thru Protocol that is used to abstract access to a SCSI channel that is produced by a SCSI host controller.
14. Protocols — USB Support	Defines USB Bus Drivers and USB Device Drivers. The protocols described include the USB Host Controller Protocol and the USB I/O Protocol.
15. Protocols — Network Support	Defines the protocols that provide access to network devices while executing in the EFI boot services environment. These protocols include the Simple Network Protocol, the PXE Base Code Protocol, and the Boot Integrity services (BIS) Protocol.
16. Protocols — Debugger Support	An optional set of protocols that provide the services required to implement a source level debugger for the EFI environment. The EFI DebugPort Protocol provides services to communicate with a remote debug host. The Debug Support Protocol provides services to hook processor exceptions, save the processor context, and restore the processor context. These protocols can be used in the implementation of a debug agent on the target system that interacts with the remote debug host.
17. Protocols — Compression Algorithm Specification	Describes in detail the EFI compression/decompression algorithm, as well as the EFI Decompress Protocol. The EFI Decompress Protocol provides a standard decompression interface for use at boot time. The EFI Decompress Protocol is used by a PCI Bus Driver to decompress EFI drivers stored in PCI Option ROMs.
18. Protocols — Device I/O Protocol	Defines the Device I/O protocol, which is used by code running in the EFI boot services environment to access memory and I/O.

continued

Table 1-1. Organization of the EFI Specification (continued)

Chapter/Appendix	**Description**
19. EFI Byte Code Virtual Machine	Defines the EFI Byte Code virtual processor and its instruction set. It also defines how EBC object files are loaded into memory, and the mechanism for transitioning from native code to EBC code and back to native code. The information in this document is sufficient to implement an EFI Byte Code interpreter, an EFI Byte Code compiler, and an EFI Byte Code linker.
A. GUID and Time Formats	Explains format of EFI GUIDs (Guaranteed Unique Identifiers).
B. Console	Describes the requirements for a basic text-based console required by EFI-conformant systems to provide communication capabilities.
C. Device Path Examples	Examples of use of the data structures that defines various hardware devices to the EFI boot services.
D. Status Codes	Lists success, error, and warning codes returned by EFI interfaces.
E. 32/64-Bit UNDI Specification	This appendix defines the 32/64-bit H/W and S/W Universal Network Driver Interfaces (UNDIs).
F. Using the Simple Pointer Protocol	This appendix provides the suggested usage of the Simple Pointer Protocol.
G. Using the EFI SCSI Pass Thru Protocol	This appendix provides an example on how the SCSI Pass Thru Protocol can be used.
H. Compression Source Code	The C source code to an implementation of the EFI Compression Algorithm.
I. Decompression Source Code	The C source code to an implementation of the EFI Decompression Algorithm.
J. EFI Byte Code Virtual Machine Opcode Summary	A summary of the opcodes in the instruction set of the EFI Byte Code Virtual Machine.
K. Alphabetic Function List	Lists all EFI interface functions alphabetically.
References	Lists all necessary and/or useful specifications, web sites, and other documentation that is referenced in this EFI Specification.
Glossary	Briefly describes terms defined or referenced by this specification.
Index	Provides an index to the key terms and concepts in the specification.

1.3 Goals

The “PC-AT” boot environment presents significant challenges to innovation within the industry. Each new platform capability or hardware innovation requires firmware developers to craft increasingly complex solutions, and often requires OS developers to make changes to their boot code before customers can benefit from the innovation. This can be a time-consuming process requiring a significant investment of resources.

The primary goal of the EFI specification is to define an alternative boot environment that can alleviate some of these considerations. In this goal, the specification is similar to other existing boot specifications. The main properties of this specification and similar solutions can be summarized by these attributes:

- *Coherent, scalable platform environment*. The specification defines a complete solution for the firmware to completely describe platform features and surface platform capabilities to the OS during the boot process. The definitions are rich enough to cover the full range of contemporary Intel architecture-based system designs.
- *Abstraction of the OS from the firmware*. The specification defines interfaces to platform capabilities. Through the use of abstract interfaces, the specification allows the OS loader to be constructed with far less knowledge of the platform and firmware that underlie those interfaces. The interfaces represent a well-defined and stable boundary between the underlying platform and firmware implementation and the OS loader. Such a boundary allows the underlying firmware and the OS loader to change provided both limit their interactions to the defined interfaces.
- *Reasonable device abstraction free of legacy interfaces*. “PC-AT” BIOS interfaces require the OS loader to have specific knowledge of the workings of certain hardware devices. This specification provides OS loader developers with something different—abstract interfaces that make it possible to build code that works on a range of underlying hardware devices without having explicit knowledge of the specifics for each device in the range.
- *Abstraction of Option ROMs from the firmware*. This specification defines interfaces to platform capabilities including standard bus types such as PCI, USB, and SCSI. The list of supported bus types may grow over time, so a mechanism to extend to future bus types is included. These defined interfaces and the ability to extend to future bus types are components of the *EFI Driver Model*. One purpose of the *EFI Driver Model* is to solve a wide range of issues that are present in existing “PC-AT” option ROMs. Like OS loaders, drivers use the abstract interfaces so device drivers and bus drivers can be constructed with far less knowledge of the platform and firmware that underlie those interfaces.
- *Architecturally shareable system partition*. Initiatives to expand platform capabilities and add new devices often require software support. In many cases, when these platform innovations are activated before the OS takes control of the platform, they must be supported by code that is specific to the platform rather than to the customer’s choice of OS. The traditional approach to this problem has been to embed code in the platform during manufacturing (for example, in flash memory devices). Demand for such persistent storage is increasing at a rapid rate. This specification defines persistent store on large mass storage media types for use by platform support code extensions to supplement the traditional approach. The definition of how this works is made clear in the

specification to ensure that firmware developers, OEMs, operating system vendors, and perhaps even third parties can share the space safely while adding to platform capability.

Defining a boot environment that delivers these attributes could be accomplished in many ways. Indeed several alternatives, perhaps viable from an academic point of view, already existed at the time this specification was written. These alternatives, however, typically presented high barriers to entry given the current infrastructure capabilities surrounding Intel architecture platforms. This specification is intended to deliver the attributes listed above while also recognizing the unique needs of an industry that has considerable investment in compatibility and a large installed base of systems that cannot be abandoned summarily. These needs drive the requirements for the additional attributes embodied in this specification:

- *Evolutionary, not revolutionary*. The interfaces and structures in the specification are designed to reduce the burden of an initial implementation as much as possible. While care has been taken to ensure that appropriate abstractions are maintained in the interfaces themselves, the design also ensures that reuse of BIOS code to implement the interfaces is possible with a minimum of additional coding effort. In other words, on IA-32 platforms the specification can be implemented initially as a thin interface layer over an underlying implementation based on existing code. At the same time, introduction of the abstract interfaces provides for migration away from legacy code in the future. Once the abstraction is established as the means for the firmware and OS loader to interact during boot, developers are free to replace legacy code underneath the abstract interfaces at leisure. A similar migration for hardware legacy is also possible. Since the abstractions hide the specifics of devices, it is possible to remove underlying hardware, and replace it with new hardware that provides improved functionality, reduced cost, or both. Clearly this requires that new platform firmware be written to support the device and present it to the OS loader via the abstract interfaces. However, without the interface abstraction, removal of the legacy device might not be possible at all.
- *Compatibility by design*. The design of the system partition structures also preserves all the structures that are currently used in the "PC-AT" boot environment. Thus it is a simple matter to construct a single system that is capable of booting a legacy OS or an EFI-aware OS from the same disk.
- *Simplifies addition of OS-neutral platform value-add*. The specification defines an open extensible interface that lends itself to the creation of platform "drivers." These may be analogous to OS drivers, providing support for new device types during the boot process, or they may be used to implement enhanced platform capabilities like fault tolerance or security. Furthermore this ability to extend platform capability is designed into the specification from the outset. This is intended to help developers avoid many of the frustrations inherent in trying to squeeze new code into the traditional BIOS environment. As a result of the inclusion of interfaces to add new protocols, OEMs or firmware developers have an infrastructure to add capability to the platform in a modular way. Such drivers may potentially be implemented using high level coding languages because of the calling conventions and environment defined in the specification. This in turn may help to reduce the difficulty and cost of innovation. The option of a system partition provides an alternative to nonvolatile memory storage for such extensions.
- *Built on existing investment.* Where possible, the specification avoids redefining interfaces and structures in areas where existing industry specifications provide adequate coverage. For example, the ACPI specification provides the OS with all the information

necessary to discover and configure platform resources. Again, this philosophical choice for the design of the specification is intended to keep barriers to its adoption as low as possible.

1.4 Target Audience

This document is intended for the following readers:

- IHVs and OEMs who will be implementing EFI drivers.
- OEMs who will be creating Intel architecture-based platforms intended to boot shrink-wrap operating systems.
- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in Intel architecture-based products.
- Operating system developers who will be adapting their shrink-wrap operating system products to run on Intel architecture-based platforms.

1.5 EFI Design Overview

The design of EFI is based on the following fundamental elements:

- *Reuse of existing table-based interfaces*. In order to preserve investment in existing infrastructure support code, both in the OS and firmware, a number of existing specifications that are commonly implemented on Intel architecture platforms must be implemented on platforms wishing to comply with the EFI specification. (See the References appendix for additional information.)
- *System partition*. The System partition defines a partition and file system that are designed to allow safe sharing between multiple vendors, and for different purposes. The ability to include a separate sharable system partition presents an opportunity to increase platform value-add without significantly growing the need for nonvolatile platform memory.
- *Boot services*. Boot services provide interfaces for devices and system functionality that can be used during boot time. Device access is abstracted through “handles” and “protocols.” This facilitates reuse of investment in existing BIOS code by keeping underlying implementation requirements out of the specification without burdening the consumer accessing the device.
- *Runtime services*. A minimal set of runtime services is presented to ensure appropriate abstraction of base platform hardware resources that may be needed by the OS during its normal operations.

Figure 1-1 shows the principal components of EFI and their relationship to platform hardware and OS software.

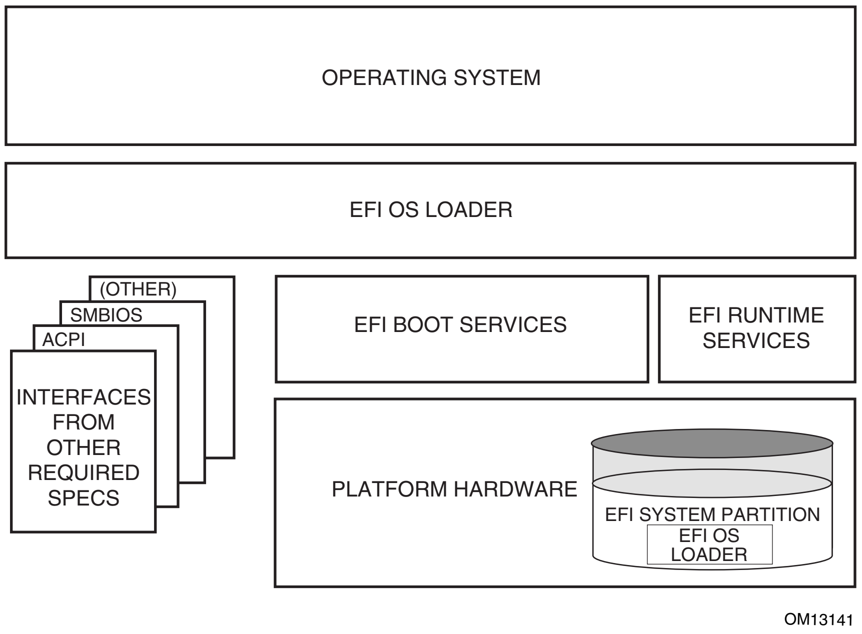


Figure 1-1. EFI Conceptual Overview

This diagram illustrates the interactions of the various components of an EFI specification-compliant system that are used to accomplish platform and OS boot.

The platform firmware is able to retrieve the OS loader image from the EFI System Partition. The specification provides for a variety of mass storage device types including disk, CD-ROM and DVD as well as remote boot via a network. Through the extensible protocol interfaces, it is possible to envision other boot media types being added, although these may require OS loader modifications if they require use of protocols other than those defined in this document

Once started, the OS loader continues to boot the complete operating system. To do so, it may use the EFI boot services and interfaces defined by this or other required specifications to survey, comprehend and initialize the various platform components and the OS software that manages them. EFI runtime services are also available to the OS loader during the boot phase.

1.6 EFI Driver Model

This section describes the goals of a driver model for EFI firmware. The goal is for this driver model to provide a mechanism for implementing bus drivers and device drivers for all types of buses and devices. At the time of writing, the bus types that must be covered include PCI, USB, SCSI, InfiniBand†, and so on.

As hardware architectures continue to evolve, the number and types of buses present in platforms are increasing. This trend is especially true in high-end servers. However, a more diverse set of bus types is being designed into desktop and mobile systems and even some embedded systems. This increasing complexity means that a simple method for describing and managing all the buses and devices in a platform is required in the preboot environment. The *EFI Driver Model* provides this simple method in the form of protocols services and boot services.

1.6.1 EFI Driver Model Goals

The *EFI Driver Model* has the following goals:

- ***Compatible*** – The *EFI Driver Model* must maintain compatibility with the *EFI 1.02 Specification*. This means that the *EFI Driver Model* must take advantage of the extensibility mechanisms in the *EFI 1.02 Specification* to add the required functionality
- ***Simple*** – Drivers written to the *EFI Driver Model* must be simple to implement and simple to maintain. The *EFI Driver Model* must allow a driver writer to concentrate on the specific device for which the driver is being developed. A driver should not be concerned with platform policy or platform management issues. These considerations should be left to the system firmware.
- ***Scalable*** – The *EFI Driver Model* must be able to adapt to all types of platforms. These platforms would include embedded systems; mobile and desktop systems, as well as workstations; and servers.
- ***Flexible*** – The *EFI Driver Model* must support the ability to enumerate all the devices, or to enumerate only those devices required to boot the required OS. The minimum device enumeration provides support for more rapid boot capability, and the full device enumeration provides the ability to perform OS installations, system maintenance, or system diagnostics on any boot device present in the system.
- ***Extensible*** – The *EFI Driver Model* must be able to extend to future bus types as they are defined.
- ***Portable*** – Drivers written to the *EFI Driver Model* must portable between platforms and between processor architectures. Initially this is limited to platforms with IA-32 family and Itanium® processors, but no processor-specific assumptions are made.
- ***Interoperable*** – Drivers must coexist with other drivers and system firmware and must do so without generating resource conflicts.
- ***Describe Complex Bus Hierarchies*** – The *EFI Driver Model* must be able to describe a variety of bus topologies from very simple single bus platforms to very complex platforms containing many buses of various types.
- ***Small Driver Footprint*** – The size of executables produced by the *EFI Driver Model* must be minimized to reduce the overall platform cost. While flexibility and extensibility are goals, the additional overhead required to support these must be kept to a minimum to prevent the size of firmware components from becoming unmanageable.
- ***Address Legacy Option ROM Issues*** – The *EFI Driver Model* must directly address and solve the constraints and limitations of legacy option ROMs. Specifically it must be possible to build add-in cards that support both EFI drivers and legacy option ROMs where such cards can execute in both legacy BIOS systems and EFI conforming platforms without modifications to the code carried on the card. The solution must provide an evolutionary path to migrate from legacy option ROMs driver to EFI drivers.

1.6.2 Legacy Option ROM Issues

This idea of supporting a driver model came from feedback on the *EFI Specification* that provided a clear, market-driven requirement for an alternative to the legacy option ROM (sometimes also referred to as an expansion ROM). The perception is that the advent of the *EFI Specification* represents a chance to escape the limitations implicit to the construction and operation of legacy option ROM images by replacing them with an alternative mechanism that works within the framework of the *EFI Specification*.

1.7 Migration Requirements

Migration requirements cover the transition period from initial implementation of this specification to a future time when all platforms and operating systems implement to this specification. During this period, two major compatibility considerations are important:

1. The ability to continue booting legacy operating systems;
   and
2. The ability to implement EFI on existing platforms by reusing as much existing firmware code to keep development resource and time requirements to a minimum.

1.7.1 Legacy Operating System Support

The EFI specification represents the preferred means for a shrink-wrap OS and firmware to communicate during the Intel architecture platform boot process. However, choosing to make a platform that complies with this specification in no way precludes a platform from also supporting existing legacy OS binaries that have no knowledge of the EFI specification.

The EFI specification does not restrict a platform designer who chooses to support both the EFI specification and a more traditional "PC-AT" boot infrastructure. If such a legacy infrastructure is to be implemented it should be developed in accordance with existing industry practice that is defined outside the scope of this specification. The choice of legacy operating systems that are supported on any given platform is left to the manufacturer of that platform.

1.7.2 Supporting the EFI Specification on a Legacy Platform

The EFI specification has been carefully designed to allow for existing systems to be extended to support it with a minimum of development effort. In particular, the abstract structures and services defined in the EFI specification can all be supported on legacy platforms.

For example, to accomplish such support on an existing IA-32 platform that uses traditional BIOS to support operating system boot, an additional layer of firmware code would need to be provided. This extra code would be required to translate existing interfaces for services and devices into support for the abstractions defined in this specification.

1.8 Conventions Used in This Document

This document uses typographic and illustrative conventions described below.

1.8.1 Data Structure Descriptions

Intel architecture processors of the IA-32 family are "little endian" machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Intel Itanium processors may be configured for both "little endian" and "big endian" operation. All implementations designed to conform to this specification will use "little endian" operation.

In some memory layout descriptions, certain fields are marked *reserved*. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

1.8.2 Protocol Descriptions

A protocol description generally has the following format:

Protocol:	The formal name of the protocol interface.
Summary:	A brief description of the protocol interface.
GUID:	The 128-bit unique identifier for the protocol interface.
Revision Number:	The revision of the protocol interface.
Protocol Interface Structure:	A "C-style" data structure definition containing the procedures and data fields produced by this protocol interface.
Parameters:	A brief description of each field in the protocol interface structure.
Related Definitions:	The type declarations and constants that are used in the protocol interface structure or any of its procedures.
Description:	A description of the functionality provided by the protocol interface including any limitations and caveats of which the caller should be aware.

1.8.3 Procedure Descriptions

A procedure description generally has the following format:

ProcedureName():	The formal name of the procedure.
Summary:	A brief description of the procedure.
Prototype:	A "C-style" procedure header defining the calling sequence.
Parameters:	The parameters defined in the template are described in further detail.
Related Definitions:	The type declarations and constants that are only used by this procedure.
Description:	A description of the functionality provided by the interface including any limitations and caveats the caller of which should be aware.
Status Codes Returned:	A description of the codes returned by the interface. Any status codes listed in this table are required to be implemented by the procedure. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

1.8.4 Instruction Descriptions

An instruction description for EBC instructions generally has the following format:

InstructionName	The formal name of the EBC Instruction.
SYNTAX:	A brief description of the EBC Instruction.
DESCRIPTION:	A description of the functionality provided by the EBC Instruction accompanied by a table that details the instruction encoding.
OPERATION:	Details the operations performed on operands.
BEHAVIORS AND RESTRICTIONS:	An item by item description of the behavior of each operand involved in the instruction and any restrictions that apply to the operands or the instruction.

1.8.5 Pseudo-Code Conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a *list* is an unordered collection of homogeneous objects. A *queue* is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be FIFO.

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the *EFI Specification*.

1.8.6 Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text (blue)	In the electronic version of this specification, any plain text underlined and in blue indicates an active link to the cross-reference.
Bold	In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.
Italic	In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
`BOLD Monospace`	Computer code, example code segments, and all prototype code segments use a **`BOLD Monospace`** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.
`BOLD Monospace`	In the electronic version of this specification, words in a **`BOLD Monospace`** typeface that is underlined and in a dark red color indicate an active hyperlink to the definition for that function or type definition. Click on the word to follow the hyperlink.

NOTE

Due to management and file size considerations, only the first occurrence of the reference on each page is an active link. Subsequent references on the same page will not be actively linked to the definition and will use the standard, nonunderlined **`BOLD Monospace`** *typeface. Find the first instance of the name (in the underlined* **`BOLD Monospace`** *typeface) on the page and click on the word to jump to the function or type definition.*

`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).

2 Overview

EFI allows the extension of platform firmware by loading EFI driver and EFI application images. When EFI drivers and EFI applications are loaded they have access to all EFI defined runtime and boot services. See Figure 2-1.

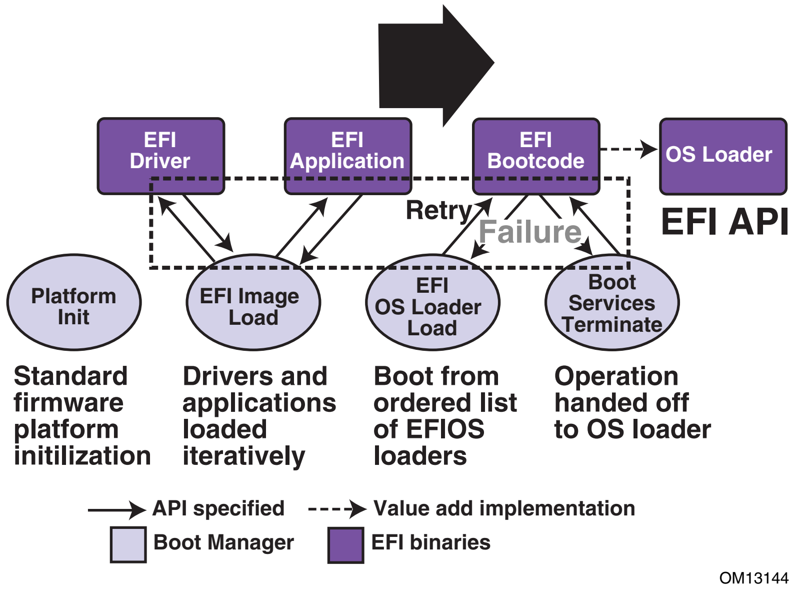


Figure 2-1. Booting Sequence

EFI allows the consolidation of boot menus from the OS loader and platform firmware into a single platform firmware menu. These platform firmware menus will allow the selection of any EFI OS loader from any partition on any boot medium that is supported by EFI boot services. An EFI OS loader can support multiple options that can appear on the user interface. It is also possible to include legacy boot options, such as booting from the A: or C: drive in the platform firmware boot menus.

EFI supports booting from media that contain an EFI OS loader or an EFI-defined System Partition. An EFI-defined System Partition is required by EFI to boot from a block device. EFI does not require any change to the first sector of a partition, so it is possible to build media that will boot on both legacy Intel architecture and EFI platforms.

2.1 Boot Manager

EFI contains a boot manager that allows the loading of EFI applications (including OS 1st stage loader) or EFI drivers from any file on an EFI defined file system or through the use of an EFI defined image loading service. EFI defines NVRAM variables that are used to point to the file to be loaded. These variables also contain application specific data that are passed directly to the EFI application. The variables also contain a human readable Unicode string that can be displayed to the user in a menu.

The variables defined by EFI allow the system firmware to contain a boot menu that can point to all the operating systems, and even multiple versions of the same operating systems. The design goal of EFI was to have one set of boot menus that could live in platform firmware. EFI only specifies the NVRAM variables used in selecting boot options. EFI leaves the implementation of the menu system as value added implementation space.

EFI greatly extends the boot flexibility of a system over the current state of the art in the PC-AT-class system. The PC-AT-class systems today are restricted to boot from the first floppy, hard drive, CD-ROM, or network card attached to the system. Booting from a common hard drive can cause lots of interoperability problems between operating systems, and different versions of operating systems from the same vendor.

2.1.1 EFI Images

EFI Images are a class of files defined by EFI that contain executable code. The most distinguishing feature of EFI Images is that the first set of bytes in the EFI Image file contains an image header that defines the encoding of the executable image.

EFI uses a subset of the PE32+ image format with a modified header signature. The modification to signature value in the PE32+ image is done to distinguish EFI images from normal PE32 executables. The "+" addition to PE32 provides the 64-bit relocation fix-up extensions to standard PE32 format.

For images with the EFI image signature, the *Subsystem* values in the PE image header are defined below. The major differences between image types are the memory type that the firmware will load the image into, and the action taken when the image's entry point exits or returns. An application image is always unloaded when control is returned from the image's entry point. A driver image is only unloaded if control is passed back with an EFI error code.

```
// PE32+ Subsystem type for EFI images
#define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION          10
#define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER  11
#define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER       12
```

The *Machine* value that is found in the PE image file header is used to indicate the machine code type of the image. The machine code types defined for images with the EFI image signature are defined below. A given platform must implement the image type native to that platform and the image type for EFI Byte Code (EBC). Support for other machine code types are optional to the platform.

```
// PE32+ Machine type for EFI images
#define EFI_IMAGE_MACHINE_IA32          0x014c
#define EFI_IMAGE_MACHINE_IA64          0x0200
#define EFI_IMAGE_MACHINE_EBC           0x0EBC
```

An EFI image is loaded into memory through the **LoadImage()** Boot Service. This service loads an image with a PE32+ format into memory. This PE32+ loader is required to load all the sections of the PE32+ image into memory. Once the image is loaded into memory, and the appropriate "fix-ups" have been performed, control is transferred to a loaded image at the *AddressOfEntryPoint* reference according to the normal indirect calling conventions of IA-32 or Itanium-based applications. All other linkage to and from an EFI image is done programmatically.

2.1.2 EFI Applications

EFI Applications are loaded by the EFI Boot Manager or by other EFI applications. To load an application the firmware allocates enough memory to hold the image, copies the sections within the application to the allocated memory and applies the relocation fix-ups needed. Once done, the allocated memory is set to be the proper type for code and data for the image. Control is then transferred to the application's entry point. When the application returns from its entry point, or when it calls the Boot Service **Exit()**, the application is unloaded from memory and control is returned to the EFI component that loaded the application.

When the EFI Boot Manager loads an application, the image handle may be used to locate the "load options" for the application. The load options are stored in nonvolatile storage and are associated with the application being loaded and executed by the EFI Boot Manager.

2.1.3 EFI OS Loaders

An EFI OS loader is a special type of EFI application that normally takes over control of the system from the EFI firmware. When loaded, the OS loader behaves like any other EFI application in that it must only use memory it has allocated from the firmware and can only use EFI services and protocols to access the devices that the firmware exposes. If the OS Loader includes any boot service style driver functions, it must use the proper EFI interfaces to obtain access to the bus specific-resources. That is, I/O and memory-mapped device registers must be accessed through the proper bus specific I/O calls like those that an EFI driver would perform.

If the OS loader experiences a problem and cannot load its operating system correctly, it can release all allocated resources and return control back to the firmware via the Boot Service **`Exit()`** call. The **`Exit()`** call allows both an error code and *`ExitData`* to be returned. The *`ExitData`* contains both a Unicode string and OS loader-specific data to be returned.

If the OS loader successfully loads its operating system, it can take control of the system by using the Boot Service **`ExitBootServices()`**. After successfully calling **`ExitBootServices()`**, all boot services in the system are terminated, including memory management, and the OS loader is responsible for the continued operation of the system.

2.1.4 EFI Drivers

EFI Drivers are loaded by the EFI Boot Manager, the EFI firmware, or by other EFI applications. To load an EFI Driver the firmware allocates enough memory to hold the image, copies the sections within the driver to the allocated memory and applies the relocation fix-ups needed. Once done, the allocated memory is set to be the proper type for code and data for the image. Control is then transferred to the driver's entry point. When the driver returns from its entry point, or when it calls the Boot Service **`Exit()`**, the driver is optionally unloaded from memory and control is returned to the EFI component that loaded the driver. A driver is not unloaded from memory if it returns a status code of **`EFI_SUCCESS`**. If the driver's return code is an error status code, then the driver is unloaded from memory.

There are two types of EFI Drivers. These are Boot Service Drivers and Runtime Drivers. The only difference between these two driver types is that Runtime Drivers are available after an OS Loader has taken control of the platform with the Boot Service **`ExitBootServices()`**. Boot Service Drivers are terminated when **`ExitBootServices()`** is called, and all the memory resources consumed by the Boot Service Drivers are released for use in the operating system environment.

2.2 Firmware Core

This section provides an overview of the services defined by EFI. These include boot services and runtime services.

2.2.1 EFI Services

The purpose of the EFI interfaces is to define a common boot environment abstraction for use by loaded EFI images, which include EFI drivers, EFI applications, and EFI OS loaders. The calls are defined with a full 64-bit interface, so that there is headroom for future growth. The goal of this set of abstracted platform calls is to allow the platform and OS to evolve and innovate independently of one another. Also, a standard set of primitive runtime services may be used by operating systems.

Platform interfaces defined in this chapter allow the use of standard Plug and Play Option ROMs as the underlying implementation methodology for the boot services. The PC industry has a huge investment in Intel Architecture Option ROM technology, and the obsolescence of this installed base of technology is not practical in the first generation of EFI-compliant systems. The interfaces have been designed in such as way as to map back into legacy interfaces. These interfaces have in no way been burdened with any restrictions inherent to legacy Option ROMs.

The EFI platform interfaces are intended to provide an abstraction between the platform and the OS that is to boot on the platform. The EFI specification also provides abstraction between diagnostics or utility programs and the platform; however, it does not attempt to implement a full diagnostic OS environment. It is envisioned that a small diagnostic OS-like environment can be easily built on top of an EFI system. Such a diagnostic environment is not described by this specification.

Interfaces added by this specification are divided into the following categories and are detailed later in this document:

- Runtime services
- Boot services interfaces, with the following subcategories:
  - Global boot service interfaces
  - Device handle-based boot service interfaces
  - Device protocols
  - Protocol services

2.2.2 Runtime Services

This section describes EFI runtime service functions. The primary purpose of the EFI runtime services is to abstract minor parts of the hardware implementation of the platform from the OS. EFI runtime service functions are available during the boot process and also at runtime provided the OS switches into flat physical addressing mode to make the runtime call. However, if the OS loader or OS uses the Runtime Service `SetVirtualAddressMap()` service, the OS will only be able to call EFI runtime services in a virtual addressing mode. All runtime interfaces are nonblocking interfaces and can be called with interrupts disabled if desired.

In all cases memory used by the EFI runtime services must be reserved and not used by the OS. EFI runtime services memory is always available to an EFI function and will never be directly manipulated by the OS or its components. EFI is responsible for defining the hardware resources used by runtime services, so the OS can synchronize with those resources when runtime service calls are made, or guarantee that the OS never uses those resources.

Table 2-1 lists the Runtime Services functions.

Table 2-1. EFI Runtime Services

Name	Description
`GetTime()`	Returns the current time, time context, and time keeping capabilities.
`SetTime()`	Sets the current time and time context.
`GetWakeupTime()`	Returns the current wakeup alarm settings.
`SetWakeupTime()`	Sets the current wakeup alarm settings.
`GetVariable()`	Returns the value of a named variable.
`GetNextVariableName()`	Enumerates variable names.
`SetVariable()`	Sets, and if needed creates, a variable.
`SetVirtualAddressMap()`	Switches all runtime functions from physical to virtual addressing.
`ConvertPointer()`	Used to convert a pointer from physical to virtual addressing.
`GetNextHighMonotonicCount()`	Subsumes the platform's monotonic counter functionality.
`ResetSystem()`	Resets all processors and devices and reboots the system.

2.3 Calling Conventions

Unless otherwise stated, all functions defined in the EFI specification are called through pointers in common, architecturally defined, calling conventions found in C compilers. Pointers to the various global EFI functions are found in the **EFI_RUNTIME_SERVICES** and **EFI_BOOT_SERVICES** tables that are located via the EFI system table. Pointers to other functions defined in this specification are located dynamically through device handles. In all cases, all pointers to EFI functions are cast with the word **EFIAPI**. This allows the compiler for each architecture to supply the proper compiler keywords to achieve the needed calling conventions. When passing pointer arguments to Boot Services, Runtime Services, and Protocol Interfaces, the caller has the following responsibilities:

1. It is the caller's responsibility to pass pointer parameters that reference physical memory locations. If a pointer is passed that does not point to a physical memory location(i.e. a memory mapped I/O region), the results are unpredictable and the system may halt.
2. It is the caller's responsibility to pass pointer parameters with correct alignment. If an unaligned pointer is passed to a function, the results are unpredictable and the system may halt.
3. It is the caller's responsibility to not pass in a **NULL** parameter to a function unless it is explicitly allowed. If a **NULL** pointer is passed to a function, the results are unpredictable and the system may hang.

Calling conventions for IA-32 or Itanium-based applications are described in more detail below. Any function or protocol may return any valid return code.

2.3.1 Data Types

Table 2-2 lists the common data types that are used in the interface definitions, and Table 2-3 lists their modifiers. Unless otherwise specified all data types are naturally aligned. Structures are aligned on boundaries equal to the largest internal datum of the structure and internal data are implicitly padded to achieve natural alignment.

Table 2-2. Common EFI Data Types

Mnemonic	Description
BOOLEAN	Logical Boolean. 1-byte value containing a 0 for **FALSE** or a 1 for **TRUE**. Other values are undefined.
INTN	Signed value of native width. (4 bytes on IA-32, 8 bytes on Itanium processor instructions)
UINTN	Unsigned value of native width. (4 bytes on IA-32, 8 bytes on Itanium processor instructions)
INT8	1-byte signed value.
UINT8	1-byte unsigned value.
INT16	2-byte signed value.
UINT16	2-byte unsigned value.

continued

Table 2-2. Common EFI Data Types (continued)

Mnemonic	**Description**
INT32	4-byte signed value.
UINT32	4-byte unsigned value.
INT64	8-byte signed value.
UINT64	8-byte unsigned value.
CHAR8	1-byte Character.
CHAR16	2-byte Character. Unless otherwise specified all strings are stored in the UTF-16 encoding format as defined by Unicode 2.1 and ISO/IEC 10646 standards.
VOID	Undeclared type.
EFI_GUID	128-bit buffer containing a unique identifier value. Unless otherwise specified, aligned on a 64-bit boundary.
EFI_STATUS	Status code. Type INTN.
EFI_HANDLE	A collection of related interfaces. Type VOID *.
EFI_EVENT	Handle to an event structure. Type VOID *.
EFI_LBA	Logical block address. Type UINT64.
EFI_TPL	Task priority level. Type UINTN.
EFI_MAC_ADDRESS	32-byte buffer containing a network Media Access Control address.
EFI_IPv4_ADDRESS	4-byte buffer. An IPv4 internet protocol address.
EFI_IPv6_ADDRESS	16-byte buffer. An IPv6 internet protocol address.
EFI_IP_ADDRESS	16-byte buffer aligned on a 4-byte boundary. An IPv4 or IPv6 internet protocol address.
<Enumerated Type>	Element of a standard ANSI C **enum** type declaration. Type INTN.

Table 2-3. Modifiers for Common EFI Data Types

Mnemonic	**Description**
IN	Datum is passed to the function.
OUT	Datum is returned from the function.
OPTIONAL	Passing the datum to the function is optional, and a **NULL** may be passed if the value is not supplied.
EFIAPI	Defines the calling convention for EFI interfaces.

2.3.2 IA-32 Platforms

All functions are called with the C language calling convention. The general-purpose registers that are volatile across function calls are **eax**, **ecx**, and **edx**. All other general-purpose registers are nonvolatile and are preserved by the target function. In addition, unless otherwise specified by the function definition, all other registers are preserved. For example, this would include the entire floating point and Intel® MMX™ technology state.

During boot services time the processor is in the following execution mode:

- Uniprocessor
- Protected mode
- Paging mode not enabled
- Selectors are set to be flat and are otherwise not used
- Interrupts are enabled–though no interrupt services are supported other than the EFI boot services timer functions (All loaded device drivers are serviced synchronously by "polling.")
- Direction flag in EFLAGs is clear
- Other general purpose flag registers are undefined
- 128 KB, or more, of available stack space

For an operating system to use any EFI runtime services, it must:

- Preserve all memory in the memory map marked as runtime code and runtime data
- Call the runtime service functions, with the following conditions:
  - — Called from the boot processor
  - — In protected mode
  - — Paging *not* enabled
  - — All selectors set to be flat with virtual = physical address. If the OS Loader or OS used **`SetVirtualAddressMap()`** to relocate the runtime services in a virtual address space, then this condition does not have to be met.
  - — Direction flag in EFLAGs clear
  - — 4 KB, or more, of available stack space
  - — Interrupts disabled
- Synchronize processor access to the legacy CMOS registers (if there are multiple processors). Only one processor can access the registers at any given time.
- ACPI Tables loaded at boot time must be contained in memory of type **`EfiACPIReclaimMemory`**.
- The system firmware must not request a virtual mapping for any memory descriptor of type **`EfiACPIReclaimMemory`** or **`EfiACPIMemoryNVS`**.
- EFI memory descriptors of type **`EfiACPIReclaimMemory`** and **`EfiACPIMemoryNVS`** must be aligned on a 4 KB boundary and must be a multiple of 4 KB in size.
- Any EFI memory descriptor that requests a virtual mapping via the **`EFI_MEMORY_DESCRIPTOR`** having the **`EFI_MEMORY_RUNTIME`** bit set must be aligned on a 4 KB boundary and must be a multiple of 4 KB in size.

- An ACPI Memory Op-region must inherit cacheability attributes from the EFI memory map. If the system memory map does not contain cacheability attributes, the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes exist in the system memory map or the ACPI name space, then the region must be assumed to be noncacheable.
- ACPI tables loaded at runtime must be contained in memory of type **`EfiACPIMemoryNVS`** or **`EfiFirmareReserved`**. The cacheability attributes for ACPI tables loaded at runtime (via ACPI LoadTable) should be defined in the EFI memory map. If no information about the table location exists in the EFI memory map, the table is assumed to be noncached.

2.3.2.1 Handoff State

When an IA-32 EFI OS is loaded, the system firmware hands off control to the OS in flat 32-bit mode. All descriptors are set to their 4 GB limits so that all of memory is accessible from all segments. The address of the IDT is not defined and thus it cannot be manipulated directly during boot services.

Figure 2-2 shows the stack after *`AddressOfEntryPoint`* in the image's PE32+ header has been called on IA-32 systems. All EFI image entry points take two parameters. These are the image handle of the EFI image, and a pointer to the EFI System Table.

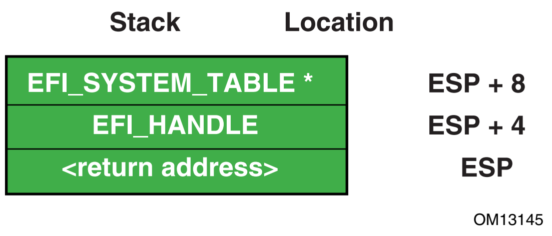


Figure 2-2. Stack after *`AddressOfEntryPoint`* Called, IA-32

2.3.3 Itanium®-Based Platforms

EFI executes as an extension to the SAL execution environment with the same rules as laid out by the SAL specification.

During boot services time the processor is in the following execution mode:

- Uniprocessor
- Physical mode
- 128 KB, or more, of available stack space
- 16 KB, or more, of available backing store space
- May only use the lower 32 floating point registers

The EFI Image may invoke both SAL and EFI procedures. Once in virtual mode, the EFI OS must switch back to physical mode to call any boot services. If **`SetVirtualAddressMap()`** has been used, then runtime service calls are made in virtual mode.

- ACPI Tables loaded at boot time must be contained in memory of type **`EfiACPIReclaimMemory`**.
- The system firmware must not request a virtual mapping for any memory descriptor of type **`EfiACPIReclaimMemory`** or **`EfiACPIMemoryNVS`**.
- EFI memory descriptors of type **`EfiACPIReclaimMemory`** and **`EfiACPIMemoryNVS`** must be aligned on an 8 KB boundary and must be a multiple of 8 KB in size.
- Any EFI memory descriptor that requests a virtual mapping via the **`EFI_MEMORY_DESCRIPTOR`** having the **`EFI_MEMORY_RUNTIME`** bit set must be aligned on a 8 KB boundary and must be a multiple of 8 KB in size.
- An ACPI Memory Op-region must inherit cacheability attributes from the EFI memory map. If the system memory map does not contain cacheability attributes the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes exist in the system memory map or the ACPI name space, then the region must be assumed to be noncacheable.
- ACPI tables loaded at runtime must be contained in memory of type **`EfiACPIMemoryNVS`** or **`EfiFirmareReserved`**. The cacheability attributes for ACPI tables loaded at runtime (via ACPI LoadTable) should be defined in the EFI memory map. If no information about the table location exists in the EFI memory map, the table is assumed to be noncached.

Refer to the *IA-64 System Abstraction Layer Specification* (see the References appendix) for details.

EFI procedures are invoked using the P64 C calling conventions defined for Itanium-based applications. Refer to the document *64 Bit Runtime Architecture and Software Conventions for IA-64* (see the References appendix) for more information.

2.3.3.1 Handoff State

EFI uses the standard P64 C calling conventions that are defined for Itanium-based operating systems. Figure 2-3 shows the stack after **ImageEntryPoint** has been called on Itanium-based systems. The arguments are also stored in registers: **out0** contains **EFI_HANDLE** and **out1** contains the address of the **EFI_SYSTEM_TABLE**. The **gp** for the EFI Image will have been loaded from the *plabel* pointed to by the *AddressOfEntryPoint* in the image's PE32+ header. All EFI image entry points take two parameters. These are the image handle of the EFI image, and a pointer to the EFI System Table.

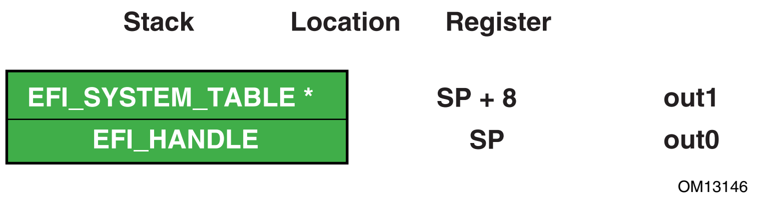


Figure 2-3. Stack after *AddressOfEntryPoint* Called, Itanium-based Systems

The SAL specification (see the References appendix) defines the state of the system registers at boot handoff. The SAL specification also defines which system registers can only be used after EFI boot services have been properly terminated.

2.4 Protocols

The protocols that a device handle supports are discovered through the **HandleProtocol()** Boot Service or the **OpenProtocol()** Boot Service. Each protocol has a specification that includes the following:

- The protocol's globally unique ID (GUID)
- The Protocol Interface structure
- The Protocol Services

To determine if the handle supports any given protocol, the protocol's GUID is passed to **HandleProtocol()** or **OpenProtocol()**. If the device supports the requested protocol, a pointer to the defined Protocol Interface structure is returned. The Protocol Interface structure links the caller to the protocol-specific services to use for this device.

Figure 2-4 shows the construction of a protocol. The EFI driver contains functions specific to one or more protocol implementations, and registers them with the Boot Service `InstallProtocolInterface()`. The firmware returns the Protocol Interface for the protocol that is then used to invoke the protocol specific services. The EFI driver keeps private, device-specific context with protocol interfaces.

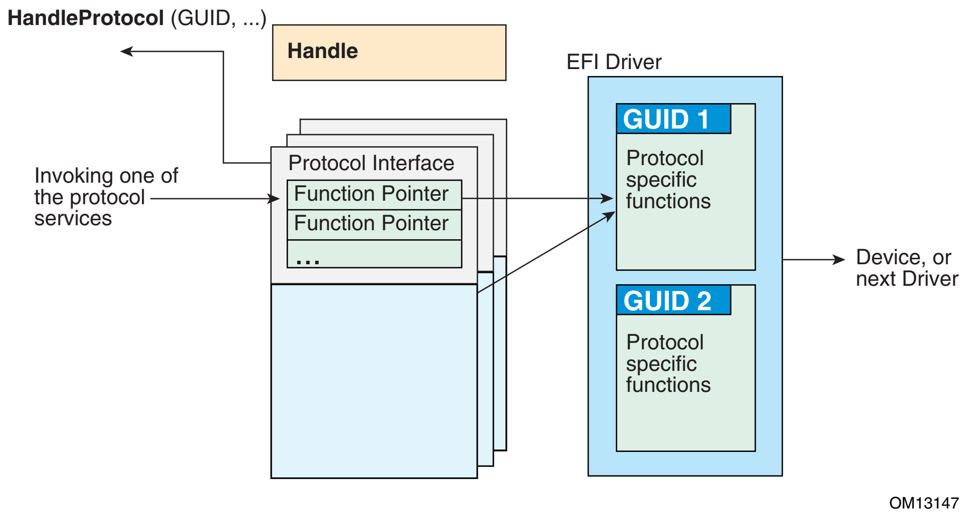

HandleProtocol (GUID, ...)
Handle
EFI Driver
Invoking one of the protocol services
Protocol Interface
Function Pointer
Function Pointer
...
GUID 1
Protocol specific functions
GUID 2
Protocol specific functions
Device, or next Driver
OM13147

Figure 2-4. Construction of a Protocol

The following C code fragment illustrates the use of protocols:

```
// There is a global "EffectsDevice" structure.  This
// structure contains information pertinent to the device.

// Connect to the ILLUSTRATION_PROTOCOL on the EffectsDevice,
// by calling HandleProtocol with the device's EFI device handle
// and the ILLUSTRATION_PROTOCOL GUID.

EffectsDevice.Handle = DeviceHandle;
Status = HandleProtocol (
             EffectsDevice.EFIHandle,
             &IllustrationProtocolGuid,
             &EffectsDevice.IllustrationProtocol
             );

// Use the EffectsDevice illustration protocol's "MakeEffects"
// service to make flashy and noisy effects.

Status = EffectsDevice.IllustrationProtocol->MakeEffects (
             EffectsDevice.IllustrationProtocol,
             TheFlashyAndNoisyEffect
             );
```

Table 2-4 lists the EFI protocols defined by this specification.

Table 2-4. EFI Protocols

Protocol	Description
LOADED_IMAGE	Provides information on the image.
DEVICE_PATH	Provides the location of the device.
DRIVER_BINDING	Provides services to determine if an EFI driver supports a given controller, and services to start and stop a given controller.
PLATFORM_DRIVER_OVERRIDE	Provide a platform specific override mechanism for the selection of the best driver for a given controller.
BUS_SPECIFIC_DRIVER_OVERRIDE	Provides a bus specific override mechanism for the selection of the best driver for a given controller.
DRIVER_CONFIGURATION	Provides user configuration options for EFI drivers and the controllers that the EFI drivers are managing.
DRIVER_DIAGNOSTICS	Provides diagnostics services for the controllers that EFI drivers are managing.
COMPONENT_NAME	Provides human readable names for EFI Drivers and the controllers that the EFI drivers are managing.
SIMPLE_INPUT	Protocol interfaces for devices that support simple console style text input.
SIMPLE_TEXT_OUTPUT	Protocol interfaces for devices that support console style text displaying.
UGA_DRAW	Protocol interfaces for devices that support graphical output.
SIMPLE_POINTER	Protocol interfaces for devices such as mice and trackballs.
SERIAL_IO	Protocol interfaces for devices that support serial character transfer.
LOAD_FILE	Protocol interface for reading a file from an arbitrary device.
SIMPLE_FILE_SYSTEM	Protocol interfaces for opening disk volume containing an EFI file system.
FILE_HANDLE	Provides access to supported file systems.
DISK_IO	A protocol interface that layers onto any BLOCK_IO interface.
BLOCK_IO	Protocol interfaces for devices that support block I/O style accesses.
UNICODE_COLLATION	Protocol interfaces for Unicode string comparison operations.
PCI_ROOT_BRIDGE_IO	Protocol interfaces to abstract memory, I/O, PCI configuration, and DMA accesses to a PCI root bridge controller.
PCI_IO	Protocol interfaces to abstract memory, I/O, PCI configuration, and DMA accesses to a PCI controller on a PCI bus.

continued

Table 2-4. EFI Protocols (continued)

Protocol	Description
SCSI_PASS_THRU	Protocol interfaces for a SCSI channel that allow SCSI Request Packets to be sent to SCSI devices.
USB_HC	Protocol interfaces to abstract access to a USB Host Controller.
USB_IO	Protocol interfaces to abstract access to a USB controller.
SIMPLE_NETWORK	Provides interface for devices that support packet based transfers.
PXE_BC	Protocol interfaces for devices that support network booting.
BIS	Protocol interfaces to validate boot images before they are loaded and invoked.
DEBUG_SUPPORT	Protocol interfaces to save and restore processor context and hook processor exceptions.
DEBUG_PORT	Protocol interface that abstracts a byte stream connection between a debug host and a debug target system.
DECOMPRESS	Protocol interfaces to decompress an image that was compressed using the EFI Compression Algorithm.
DEVICE_IO	Protocol interfaces for performing device I/O.
EBC	Protocols interfaces required to support an EFI Byte Code interpreter.

2.5 EFI Driver Model

The *EFI Driver Model* is intended to simplify the design and implementation of device drivers, and produce small executable image sizes. As a result, some complexity has been moved into bus drivers and in a larger part into common firmware services.

A device driver is required to produce a Driver Binding Protocol on the same image handle on which the driver was loaded. It then waits for the system firmware to connect the driver to a controller. When that occurs, the device driver is responsible for producing a protocol on the controller's device handle that abstracts the I/O operations that the controller supports. A bus driver performs these exact same tasks. In addition, a bus driver is also responsible for discovering any child controllers on the bus, and creating a device handle for each child controller found.

One assumption is that the architecture of a system can be viewed as a set of one or more processors connected to one or more core chipsets. The core chipsets are responsible for producing one or more I/O buses. The *EFI Driver Model* does not attempt to describe the processors or the core chipsets. Instead, the *EFI Driver Model* describes the set of I/O buses produced by the core chipsets, and any children of these I/O buses. These children can either be devices or additional I/O buses. This can be viewed as a tree of buses and devices with the core chipsets at the root of that tree.

The leaf nodes in this tree structure are peripherals that perform some type of I/O. This could include keyboards, displays, disks, network, etc. The nonleaf nodes are the buses that move data between devices and buses, or between different bus types. Figure 2-5 shows a sample desktop system with four buses and six devices.

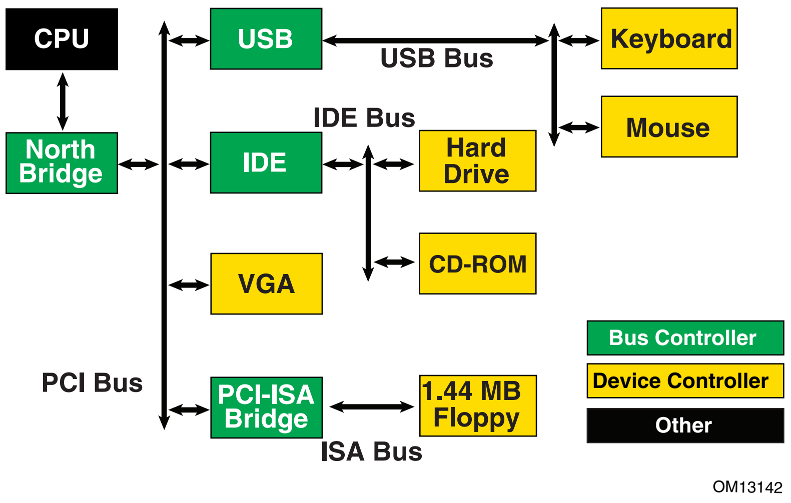


Figure 2-5. Desktop System

Figure 2-6 is an example of a more complex server system. The idea is to make the *EFI Driver Model* simple and extensible so more complex systems like the one below can be described and managed in the preboot environment. This system contains six buses and eight devices.

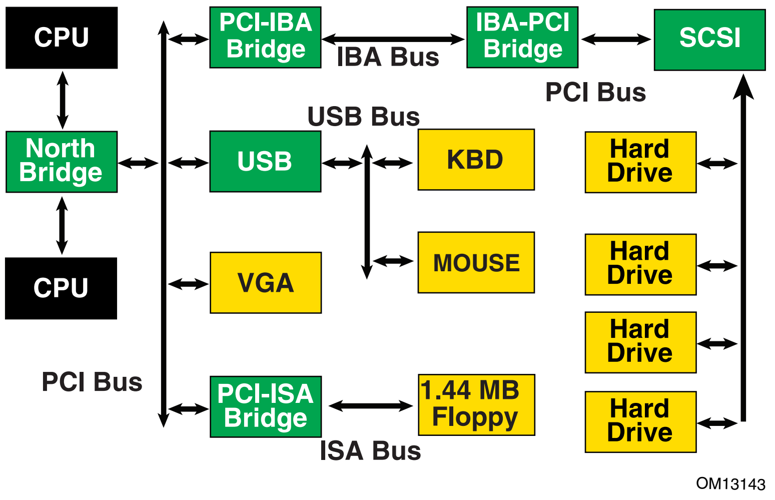


Figure 2-6. Server System

The combination of firmware services, bus drivers, and device drivers in any given platform is likely to be produced by a wide variety of vendors including OEMs, IBVs, and IHVs. These different components from different vendors are required to work together to produce a protocol for an I/O device than can be used to boot an EFI compliant operating system. As a result, the *EFI Driver Model* is described in great detail in order to increase the interoperability of these components.

This remainder of this section is a brief overview of the *EFI Driver Model*. It describes the legacy option ROM issues that the *EFI Driver Model* is designed to address, the entry point of a driver, host bus controllers, properties of device drivers, properties of bus drivers, and how the *EFI Driver Model* can accommodate hot-plug events.

2.5.1 Legacy Option ROM Issues

Legacy option ROMs have a number of constraints and limitations that restrict innovation on the part of platform designers and adapter vendors. At the time of writing, both ISA and PCI adapters use legacy option ROMs. For the purposes of this discussion, only PCI option ROMs will be considered; legacy ISA option ROMs are not supported as part of the *EFI Specification*.

The following is a list of the major constraints and limitations of legacy option ROMs. For each issue, the design considerations that went into the design of the *EFI Driver Model* are also listed. Thus, the design of the *EFI Driver Model* directly addresses the requirements for a solution to overcome the limitations implicit to PC-AT-style legacy option ROMs.

2.5.1.1 IA-32 16-Bit Real Mode Binaries

Legacy option ROMs typically contain 16-bit real mode code for an IA-32 processor. This means that the legacy option ROM on a PCI card cannot be used in platforms that do not support the execution of IA-32 real mode binaries. Also, 16-bit real mode only allows the driver to access directly the lower 1 MB of system memory. It is possible for the driver to switch the processor into modes other than real mode in order to access resources above 1 MB, but this requires a lot of additional code, and causes interoperability issues with other option ROMs and the system BIOS. Also, option ROMs that switch the processor into to alternate execution modes are not compatible with Itanium Processors.

EFI Driver Model design considerations:

- Drivers need flat memory mode with full access to system components.
- Drivers need to be written in C so they are portable between processor architectures.
- Drivers may be compiled into a virtual machine executable, allowing a single binary driver to work on machines using different processor architectures.

2.5.1.2 Fixed Resources for Working with Option ROMs

Since legacy option ROMs can only directly address the lower 1 MB of system memory, this means that the code from the legacy option ROM must exist below 1 MB. In a PC-AT platform, memory from 0x00000-0x9FFFF is system memory. Memory from 0xA0000-0xBFFFF is VGA memory, and memory from 0xF0000-0xFFFFF is reserved for the system BIOS. Also, since system BIOS has become more complex over the years, many platforms also use 0xE0000-0xEFFFF for system BIOS. This leaves 128 KB of memory from 0xC0000-0xDFFFF for legacy option ROMs. This limits how many legacy option ROMs can be run during BIOS POST.

Also, it is not easy for legacy option ROMs to allocate system memory. Their choices are to allocate memory from Extended BIOS Data Area (EBDA), allocate memory through a Post Memory Manager (PMM), or search for free memory based on a heuristic. Of these, only EBDA is standard, and the others are not used consistently between adapters, or between BIOS vendors, which adds complexity and the potential for conflicts.

EFI Driver Model design considerations:

- Drivers need flat memory mode with full access to system components.
- Drivers need to be relocatable, so they can be loaded anywhere in memory (PE/COFF Images)
- Drivers should allocate memory through the EFI boot services. These are well-specified interfaces, and can be guaranteed to function as expected across a wide variety of platform implementations.

2.5.1.3 Matching Option ROMs to their Devices

It is not clear which controller may be managed by a particular legacy option ROM. Some legacy option ROMs search the entire system for controllers to manage. This can be a lengthy process depending on the size and complexity of the platform. Also, due to limitation in BIOS design, all the legacy option ROMs must be executed, and they must scan for all the peripheral devices before an operating system can be booted. This can also be a lengthy process, especially if SCSI buses must be scanned for SCSI devices. This means that legacy option ROMs are making policy decision about how the platform is being initialized, and which controllers are managed by which legacy option ROMs. This makes it very difficult for a system designer to predict how legacy option ROMs will interact with each other. This can also cause issues with on-board controllers, because a legacy option ROM may incorrectly choose to manage the on-board controller.

EFI Driver Model design considerations:

- Driver to controller matching must be deterministic
- Give OEMs more control through Platform Driver Override Protocol and Driver Configuration Protocol
- It must be possible to start only the drivers and controllers required to boot an operating system.

2.5.1.4 Ties to PC-AT System Design

Legacy option ROMs assume a PC-AT-like system architecture. Many of them include code that directly touches hardware registers. This can make them incompatible on legacy-free and headless platforms. Legacy option ROMs may also contain setup programs that assume a PC-AT-like system architecture to interact with a keyboard or video display. This makes the setup application incompatible on legacy-free and headless platforms.

EFI Driver Model design considerations:

- Drivers should use well-defined protocols to interact with system hardware, system input devices, and system output devices.

2.5.1.5 Ambiguities in Specification and Workarounds Born of Experience

Many legacy option ROMs and BIOS code contain workarounds because of incompatibilities between legacy option ROMs and system BIOS. These incompatibilities exist in part because there are no clear specifications on how to write a legacy option ROM or write a system BIOS.

Also, interrupt chaining and boot device selection is very complex in legacy option ROMs. It is not always clear which device will be the boot device for the OS.

EFI Driver Model design considerations:

- EFI Drivers and EFI firmware is written to follow the EFI *Specification.* Since both components have a clearly defined specification, compliance tests can be developed to prove that drivers and system firmware are compliant. This should eliminate the need to build workarounds into either drivers or system firmware (other than those that might be required to address specific hardware issues).
- Give OEMs more control through Platform Driver Override Protocol and Driver Configuration Protocol and other OEM value-add components to manage the boot device selection process.

2.5.2 Driver Initialization

The file for a driver image must be loaded from some type of media. This could include ROM, FLASH, hard drives, floppy drives, CD-ROM, or even a network connection. Once a driver image has been found, it can be loaded into system memory with the boot service **`LoadImage()`**. **`LoadImage()`** loads a PE/COFF formatted image into system memory. A handle is created for the driver, and a Loaded Image Protocol instance is placed on that handle. A handle that contains a Loaded Image Protocol instance is called an *Image Handle*. At this point, the driver has not been started. It is just sitting in memory waiting to be started. Figure 2-7 shows the state of an image handle for a driver after **`LoadImage()`** has been called.

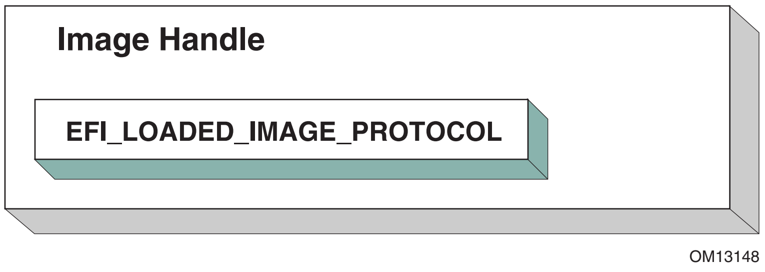


Figure 2-7. Image Handle

After a driver has been loaded with the boot service **`LoadImage()`**, it must be started with the boot service **`StartImage()`**. This is true of all types of EFI Applications and EFI Drivers that can be loaded and started on an EFI-compliant system. The entry point for a driver that follows the *EFI Driver Model* must follow some strict rules. First, it is not allowed to touch any hardware. Instead, the driver is only allowed to install protocol instances onto its own *Image Handle.* A driver that follows the *EFI Driver Model* is *required* to install an instance of the Driver Binding Protocol onto its own *Image Handle.* It may optionally install the Driver Configuration Protocol, the Driver Diagnostics Protocol, or the Component Name Protocol. In addition, if a driver wishes to be unloadable it may optionally update the Loaded Image Protocol to provide its own **`Unload()`** function. Finally, if a driver needs to perform any special operations when the boot service **`ExitBootServices()`** is called, it may optionally create an event with a notification function that is triggered when the boot service **`ExitBootServices()`** is called. An *Image Handle* that contains a Driver Binding Protocol instance is known as a *Driver Image Handle*. Figure 2-8 shows a possible configuration for the *Image Handle* from Figure 2-7 after the boot service **`StartImage()`** has been called.

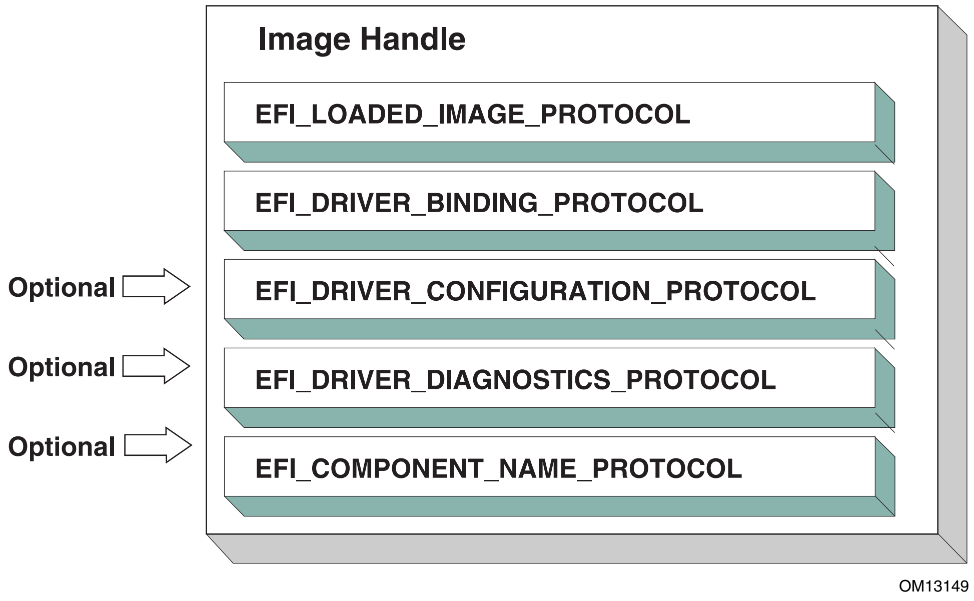


Image Handle
EFI_LOADED_IMAGE_PROTOCOL
EFI_DRIVER_BINDING_PROTOCOL
Optional
EFI_DRIVER_CONFIGURATION_PROTOCOL
Optional
EFI_DRIVER_DIAGNOSTICS_PROTOCOL
Optional
EFI_COMPONENT_NAME_PROTOCOL
OM13149

Figure 2-8. Driver Image Handle

2.5.3 Host Bus Controllers

Drivers are not allowed to touch any hardware in the driver's entry point. As a result, drivers will be loaded and started, but they will all be waiting to be told to manage one or more controllers in the system. A platform component, like the EFI Boot Manager, is responsible for managing the connection of drivers to controllers. However, before even the first connection can be made, there has to be some initial collection of controllers for the drivers to manage. This initial collection of controllers is known as the *Host Bus Controllers*. The I/O abstractions that the *Host Bus Controllers* provide are produced by firmware components that are outside the scope of the *EFI Driver Model*. The device handles for the *Host Bus Controllers* and the I/O abstraction for each one must be produced by the core firmware on the platform, or an EFI Driver that may not follow the *EFI Driver Model*. See the *PCI Root Bridge I/O Protocol Specification* for an example of an I/O abstraction for PCI buses.

A platform can be viewed as a set of processors and a set of core chipset components that may produce one or more host buses. Figure 2-9 shows a platform with ***n*** processors (CPUs), and a set of core chipset components that produce ***m*** host bridges.

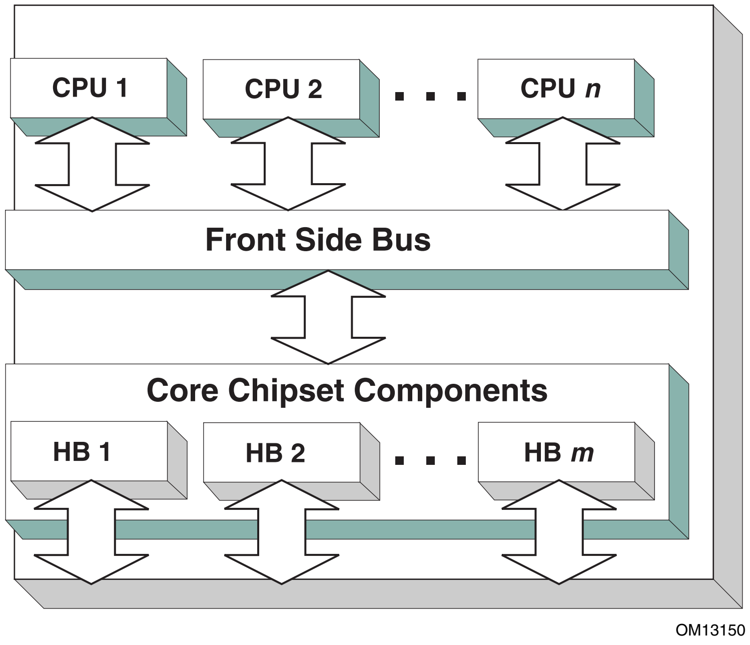


Figure 2-9. Host Bus Controllers

Each host bridge is represented in EFI as a device handle that contains a Device Path Protocol instance, and a protocol instance that abstracts the I/O operations that the host bus can perform. For example, a PCI Host Bus Controller supports one or more PCI Root Bridges that are abstracted by the PCI Root Bridge I/O Protocol. Figure 2-10 shows an example device handle for a PCI Root Bridge.

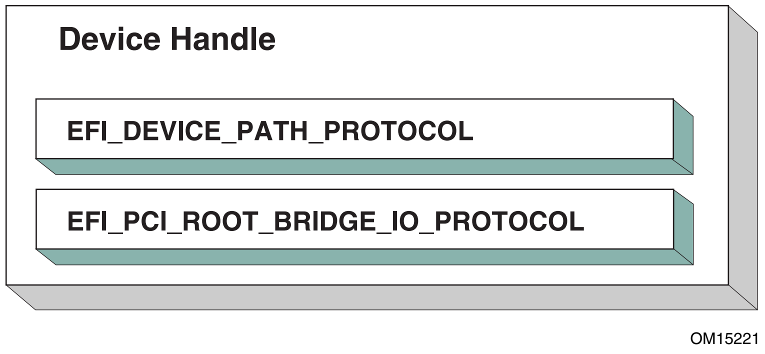


Figure 2-10. PCI Root Bridge Device Handle

A PCI Bus Driver could connect to this PCI Root Bridge, and create child handles for each of the PCI devices in the system. PCI Device Drivers should then be connected to these child handles, and produce I/O abstractions that may be used to boot an EFI compliant OS. The following section describes the different types of drivers that can be implemented within the *EFI Driver Model*. The *EFI Driver Model* is very flexible, so all the possible types of drivers will not be discussed here. Instead, the major types will be covered that can be used as a starting point for designing and implementing additional driver types.

2.5.4 Device Drivers

A device driver is not allowed to create any new device handles. Instead, it installs additional protocol interfaces on an existing device handle. The most common type of device driver will attach an I/O abstraction to a device handle that was created by a bus driver. This I/O abstraction may be used to boot an EFI compliant OS. Some example I/O abstractions would include Simple Text Output, Simple Input, Block I/O, and Simple Network Protocol. Figure 2-11 shows a device handle before and after a device driver is connected to it. In this example, the device handle is a child of the XYZ Bus, so it contains an XYZ I/O Protocol for the I/O services that the XYZ bus supports. It also contains a Device Path Protocol that was placed there by the XYZ Bus Driver. The Device Path Protocol is not required for all device handles. It is only required for device handles that represent physical devices in the system. Handles for virtual devices will not contain a Device Path Protocol.

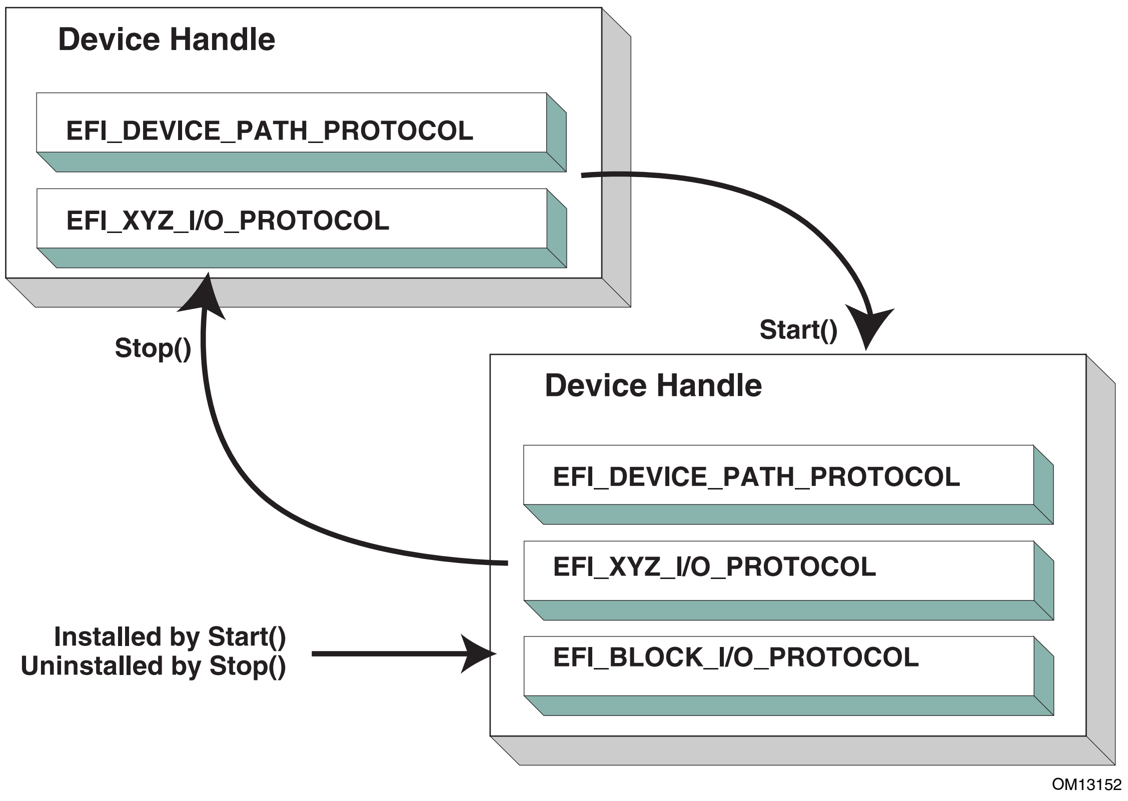


Figure 2-11. Connecting Device Drivers

The device driver that connects to the device handle in Figure 2-11 must have installed a Driver Binding Protocol on its own image handle. The Driver Binding Protocol contains three functions called `Supported()`, `Start()`, and `Stop()`. The `Supported()` function tests to see if the driver supports a given controller. In this example, the driver will check to see if the device handle supports the Device Path Protocol and the XYZ I/O Protocol. If a driver's `Supported()` function passes, then the driver can be connected to the controller by calling the driver's `Start()` function. The `Start()` function is what actually adds the additional I/O protocols to a device handle. In this example, the Block I/O Protocol is being installed. To provide symmetry, the Driver Binding Protocol also has a `Stop()` function that forces the driver to stop managing a device handle. This will cause the device driver to uninstall any protocol interfaces that were installed in `Start()`.

The `Supported()`, `Start()`, and `Stop()` functions of the EFI Driver Binding Protocol are required to make use of the boot service `OpenProtocol()` to get a protocol interface and the boot service `CloseProtocol()` to release a protocol interface. `OpenProtocol()` and `CloseProtocol()` update the handle database maintained by the system firmware to track which drivers are consuming protocol interfaces. The information in the handle database can be used to retrieve information about both drivers and controllers. The new boot service `OpenProtocolInformation()` can be used to get the list of components that are currently consuming a specific protocol interface.

2.5.5 Bus Drivers

Bus drivers and device drivers are virtually identical from the *EFI Driver Model*'s point of view. The only difference is that a bus driver creates new device handles for the child controllers that the bus driver discovers on its bus. As a result, bus drivers are slightly more complex than device drivers, but this in turn simplifies the design and implementation of device drivers. There are two major types of bus drivers. The first creates handles for all child controllers on the first call to `Start()`. The other type allows the handles for the child controllers to be created across multiple calls to `Start()`. This second type of bus driver is very useful in supporting a rapid boot capability. It allows a few child handles or even one child handle to be created. On buses that take a long time to enumerate all of their children (e.g. SCSI), this can lead to a very large timesaving in booting a platform. Figure 2-12 shows the tree structure of a bus controller before and after `Start()` is called. The dashed line coming into the bus controller node represents a link to the bus controller's parent controller. If the bus controller is a *Host Bus Controller*, then it will not have a parent controller. Nodes A, B, C ,D, and E represent the child controllers of the bus controller.

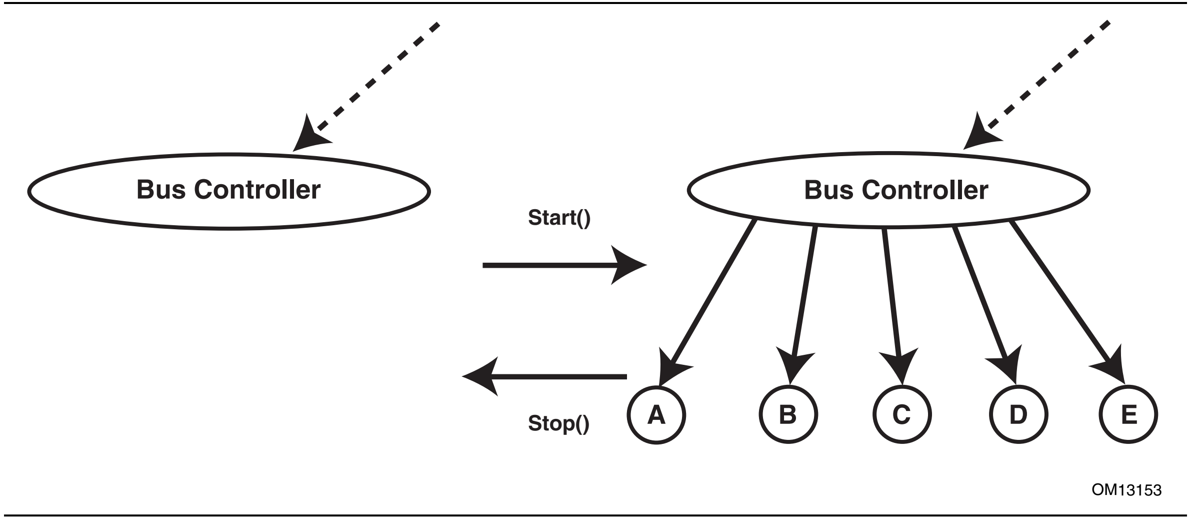


Figure 2-12. Connecting Bus Drivers

A bus driver that supports creating one child on each call to `Start()` might choose to create child C first, and then child E, and then the remaining children A, B, and D. The `Supported()`, `Start()`, and `Stop()` functions of the Driver Binding Protocol are flexible enough to allow this type of behavior.

A bus driver must install protocol interfaces onto every child handle that is creates. At a minimum, it must install a protocol interface that provides an I/O abstraction of the bus's services to the child controllers. If the bus driver creates a child handle that represents a physical device, then the bus driver must also install a Device Path Protocol instance onto the child handle. A bus driver may optionally install a Bus Specific Driver Override Protocol onto each child handle. This protocol is used when drivers are connected to the child controllers. The boot service `ConnectController()` uses architecturally defined precedence rules to choose the best set of drivers for a given controller. The Bus Specific Driver Override Protocol has higher precedence than a general driver search algorithm, and lower precedence than platform overrides. An example of a bus specific driver selection occurs with PCI. A PCI Bus Driver gives a driver stored in a PCI controller's option ROM a higher precedence than drivers stored elsewhere in the platform. Figure 2-13 shows an example child device handle that was created by the XYZ Bus Driver that supports a bus specific driver override mechanism.

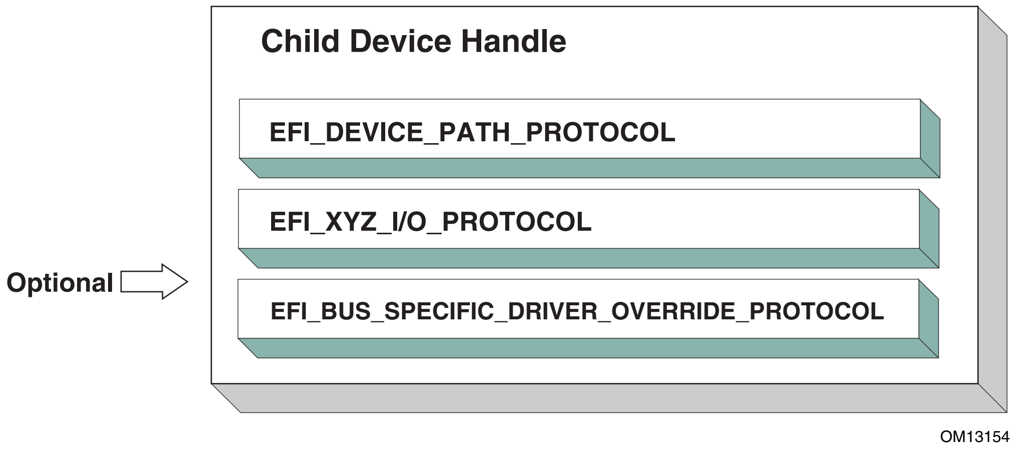


Figure 2-13. Child Device Handle with a Bus Specific Override

2.5.6 Platform Components

Under the *EFI Driver Model*, the act of connecting and disconnecting drivers from controllers in a platform is under the platform firmware's control. This will typically be implemented as part of the EFI Boot Manager, but other implementations are possible. The boot services `ConnectController()` and `DisconnectController()` can be used by the platform firmware to determine which controllers get started and which ones do not. If the platform wishes to perform system diagnostics or install an operating system, then it may choose to connect drivers to all possible boot devices. If a platform wishes to boot a preinstalled operating system, it may choose to only connect drivers to the devices that are required to boot the selected operating system. The *EFI Driver Model* supports both these modes of operation through the boot services `ConnectController()` and `DisconnectController()`. In addition, since the platform component that is in charge of booting the platform has to work with device paths for console devices and boot options, all of the services and protocols involved in the *EFI Driver Model* are optimized with device paths in mind.

Since the platform firmware may choose to only connect the devices required to produce consoles and gain access to a boot device, the OS present device drivers cannot assume that an EFI driver for a device has been executed. The presence of an EFI driver in the system firmware or in an option ROM does not guarantee that the EFI driver will be loaded, executed, or allowed to manage any devices in a platform. All OS present device drivers must be able to handle devices that have been managed by an EFI driver and devices that have not been managed by an EFI driver.

The platform may also choose to produce a protocol named the Platform Driver Override Protocol. This is similar to the Bus Specific Driver Override Protocol, but it has higher priority. This gives the platform firmware the highest priority when deciding which drivers are connected to which controllers. The Platform Driver Override Protocol is attached to a handle in the system. The boot service `ConnectController()` will make use of this protocol if it is present in the system.

2.5.7 Hot-Plug Events

In the past, system firmware has not had to deal with hot-plug events in the preboot environment. However, with the advent of buses like USB, where the end user can add and remove devices at any time, it is important to make sure that it is possible to describe these types of buses in the *EFI Driver Model*. It is up to the bus driver of a bus that supports the hot adding and removing of devices to provide support for such events. For these types of buses, some of the platform management is going to have to move into the bus drivers. For example, when a keyboard is hot added to a USB bus on a platform, the end user would expect the keyboard to be active. A USB Bus driver could detect the hot-add event and create a child handle for the keyboard device. However, because drivers are not connected to controllers unless **`ConnectController()`** is called, the keyboard would not become an active input device. Making the keyboard driver active requires the USB Bus driver to call **`ConnectController()`** when a hot-add event occurs. In addition, the USB Bus Driver would have to call **`DisconnectController()`** when a hot-remove event occurs.

Device drivers are also affected by these hot-plug events. In the case of USB, a device can be removed without any notice. This means that the **`Stop()`** functions of USB device drivers will have to deal with shutting down a driver for a device that is no longer present in the system. As a result, any outstanding I/O requests will have to be flushed without actually being able to touch the device hardware.

In general, adding support for hot-plug events greatly increases the complexity of both bus drivers and device drivers. Adding this support is up to the driver writer, so the extra complexity and size of the driver will need to be weighed against the need for the feature in the preboot environment.

2.6 Requirements

This document is an architectural specification. As such, care has been taken to specify architecture in ways that allow maximum flexibility in implementation. However, there are certain requirements on which elements of this specification must be implemented to ensure that operating system loaders and other code designed to run with EFI boot services can rely upon a consistent environment.

For the purposes of describing these requirements, the specification is broken up into required and optional elements. In general, an optional element is completely defined in the section that matches the element name. For required elements however, the definition may in a few cases not be entirely self contained in the section that is named for the particular element. In implementing required elements, care should be taken to cover all the semantics defined in this specification that relate to the particular element.

2.6.1 Required Elements

Table 2-5 lists the required elements. Any system that is designed to conform to the EFI specification *must* provide a complete implementation of all these elements. This means that all the required service functions and protocols must be present and the implementation must deliver the full semantics defined in the specification for all combinations of calls and parameters. Implementers of EFI applications, drivers or operating system loaders that are designed to run on a broad range of systems conforming to the EFI specification may assume that all such systems implement all the required elements.

A system vendor may choose not to implement all the required elements, for example on specialized system configurations that do not support all the services and functionality implied by the required elements. However, since most EFI applications, drivers and operating system loaders are written assuming all the required elements are present on a system that implements the EFI specification; any such code is likely to require explicit customization to run on a less than complete implementation of the required elements in the EFI specification.

Table 2-5. Required EFI Implementation Elements

Element	Description
EFI System Table	Provides access to EFI Boot Services, EFI Runtime Services, consoles, firmware vendor information, and the system configuration tables.
EFI Boot Services	All functions defined as boot services.
EFI Runtime Services	All functions defined as runtime services.
LOADED_IMAGE protocol	Provides information on the image.
DEVICE_PATH protocol	Provides the location of the device.
DECOMPRESS protocol	Protocol interfaces to decompress an image that was compressed using the EFI Compression Algorithm.
EBC Interpreter	An EFI Byte Code Interpreter is required so EFI images compiled to EFI Byte Code executables are guaranteed to function on all EFI compliant platforms. The EBC Interpreter must also produce the EBC protocol.

2.6.2 Platform-Specific Elements

There are a number of EFI elements that can be added or removed depending on the specific features that a platform requires. Platform firmware developers are required to implement EFI elements based upon the features included. The following is a list of potential platform features and the EFI elements that are required for each feature type:

1. If a platform includes console devices, the Simple Input Protocol and Simple Text Output Protocol must be implemented.
2. If a platform includes graphical console devices, then the UGA Draw Protocol and the UGA I/O Protocol must be implemented. In order to support UGA, a platform must contain a driver to consume UGA Draw Protocol and produce Simple Text Output Protocol even if the UGA Draw Protocol is produced by an external driver.
3. If a platform includes a pointer device as part of its console support, the Simple Pointer Protocol must be implemented.
4. If a platform includes the ability to boot from a disk device, then the Block I/O Protocol, the Disk I/O Protocol, the Simple File System Protocol, and the Unicode Collation Protocol are required. In addition, partition support for MBR, GPT, and El Torito must be implemented. An external driver may produce the Block I/O Protocol. All other protocols required to boot from a disk device must be carried as part of the platform.
5. If a platform includes the ability to boot from a network device, then the UNDI interface, the Simple Network Protocol, and the PXE Base Code Protocol are required. If a platform includes the ability to validate a boot image received through a network device, the Boot Integrity Services Protocol is also required. An external driver may produce the UNDI interface. All other protocols required to boot from a network device must be carried by the platform.
6. If a platform includes a byte-stream device such as a UART, then the Serial I/O Protocol must be implemented.
7. If a platform includes PCI bus support, then the PCI Root Bridge I/O Protocol, the PCI I/O Protocol, and the Device I/O Protocol must be implemented.
8. If a platform includes USB bus support, then the USB Host Controller Protocol and the USB I/O Protocol must be implemented. An external device can support USB by producing a USB Host Controller Protocol.
9. If a platform includes an I/O subsystem that uses SCSI command packets, the SCSI Pass Thru Protocol must be implemented.
10. If a platform includes debugging capabilities, then the Debug Support Protocol, the Debug Port Protocol, and the Debug Image Info Table must be implemented.
11. If a platform includes the ability to override the default driver to the controller matching algorithm provided by the EFI Driver Model, then the Platform Driver Override Protocol must be implemented.

2.6.3 Driver-Specific Elements

There are a number of EFI elements that can be added or removed depending on the features that a specific driver requires. Drivers can be implemented by platform firmware developers to support buses and devices in a specific platform. Drivers can also be implemented by add-in card vendors for devices that might be integrated into the platform hardware or added to a platform through an expansion slot. The following list includes possible driver features, and the EFI elements that are required for each feature type:

1. If a driver follows the EFI 1.10 Driver Model, the EFI Driver Binding Protocol must be implemented. It is strongly recommended that all drivers that follow the EFI Driver Model also implement the Component Name Protocol.
2. If a driver requires configuration information, the Driver Configuration Protocol must be implemented. A driver is not allowed to interact with the user unless the Driver Configuration Protocol is invoked.
3. If a driver requires diagnostics, the Driver Diagnostics Protocol must be implemented. In order to support low boot times, limit diagnostics during normal boots. Time consuming diagnostics should be deferred until the Driver Diagnostics Protocol is invoked.
4. If a bus supports devices that are able to provide containers for EFI drivers (e.g. option ROMs), then the bus driver for that bus type must implement the Bus Specific Driver Override Protocol.
5. If a driver is written for a console output device, then the Simple Text Output Protocol must be implemented.
6. If a driver is written for a graphical console output device, then the UGA Draw Protocol and the UGA I/O Protocol must be implemented.
7. If a driver is written for a console input device, then the Simple Input Protocol must be implemented.
8. If a driver is written for a pointer device, then the Simple Pointer Protocol must be implemented.
9. If a driver is written for a network device, then the UNDI interface must be implemented.
10. If a driver is written for a disk device, then the Block I/O Protocol must be implemented.
11. If a driver is written for a device that is not a block oriented device but one that can provide a file system-like interface, then the Simple File System Protocol must be implemented.
12. If a driver is written for a PCI root bridge, then the PCI Root Bridge I/O Protocol, the PCI I/O Protocol, and the Device I/O Protocol must be implemented.
13. If a driver is written for a USB host controller, then the USB Host Controller Protocol must be implemented.
14. If a driver is written for a SCSI controller, then the SCSI Pass Thru Protocol must be implemented.
15. If a driver is written for a boot device that is not a block-oriented device, a file system-based device, or a console device, then the Load File Protocol must be implemented.

3 Boot Manager

The EFI boot manager is a firmware policy engine that can be configured by modifying architecturally defined global NVRAM variables. The boot manager will attempt to load EFI drivers and EFI applications (including EFI OS boot loaders) in an order defined by the global NVRAM variables. The platform firmware must use the boot order specified in the global NVRAM variables for normal boot. The platform firmware may add extra boot options or remove invalid boot options from the boot order list.

The platform firmware may also implement value added features in the boot manager if an exceptional condition is discovered in the firmware boot process. One example of a value added feature would be not loading an EFI driver if booting failed the first time the driver was loaded. Another example would be booting to an OEM-defined diagnostic environment if a critical error was discovered in the boot process.

The boot sequence for EFI consists of the following:

- The boot order list is read from a globally defined NVRAM variable. The boot order list defines a list of NVRAM variables that contain information about what is to be booted. Each NVRAM variable defines a Unicode name for the boot option that can be displayed to a user.
- The variable also contains a pointer to the hardware device and to a file on that hardware device that contains the EFI image to be loaded.
- The variable might also contain paths to the OS partition and directory along with other configuration specific directories.

The NVRAM can also contain load options that are passed directly to the EFI image. The platform firmware has no knowledge of what is contained in the load options. The load options are set by higher level software when it writes to a global NVRAM variable to set the platform firmware boot policy. This information could be used to define the location of the OS kernel if it was different than the location of the EFI OS loader.

3.1 Firmware Boot Manager

The boot manager is a component in the EFI firmware that determines which EFI drivers and EFI applications should be explicitly loaded and when. Once the EFI firmware is initialized, it passes control to the boot manager. The boot manager is then responsible for determining what to load and any interactions with the user that may be required to make such a decision. Much of the behavior of the boot manager is left up to the firmware developer to decide, and details of boot manager implementation are outside the scope of this specification. In particular, likely implementation options might include any console interface concerning boot, integrated platform management of boot selections, possible knowledge of other internal applications or recovery drivers that may be integrated into the system through the boot manager.

Programmatic interaction with the boot manager is accomplished through globally defined variables. On initialization the boot manager reads the values which comprise all of the published load options among the EFI environment variables. By using the **SetVariable()** function the data that contain these environment variables can be modified.

Each load option entry resides in a *Boot####* variable or a *Driver####* variable where the *####* is replaced by a unique option number in printable hexadecimal representation using the digits 0–9, and the upper case versions of the characters A–F (0000–FFFF). The *####* must always be four digits, so small numbers must use leading zeros. The load options are then logically ordered by an array of option numbers listed in the desired order. There are two such option ordering lists. The first is *DriverOrder* that orders the *Driver####* load option variables into their load order. The second is *BootOrder* that orders the *Boot####* load options variables into their load order.

For example, to add a new boot option, a new *Boot####* variable would be added. Then the option number of the new *Boot####* variable would be added to the *BootOrder* ordered list and the *BootOrder* variable would be rewritten. To change boot option on an existing *Boot####*, only the *Boot####* variable would need to be rewritten. A similar operation would be done to add, remove, or modify the driver load list.

If the boot via *Boot####* returns with a status of **EFI_SUCCESS** the boot manager will stop processing the *BootOrder* variable and present a boot manager menu to the user. If a boot via *Boot####* returns a status other than **EFI_SUCCESS**, the boot has failed and the next *Boot####* in the *BootOrder* variable will be tried until all possibilities are exhausted.

The boot manager may perform automatic maintenance of the database variables. For example, it may remove unreferenced load option variables, any unparseable or unloadable load option variables, and rewrite any ordered list to remove any load options that do not have corresponding load option variables. In addition, the boot manager may automatically update any ordered list to place any of its own load options where it desires. The boot manager can also, at its own discretion, provide for manual maintenance operations as well. Examples include choosing the order of any or all load options, activating or deactivating load options, etc.

The boot manager is required to process the Driver load option entries before the Boot load option entries. The boot manager is also required to initiate a boot of the boot option specified by the *BootNext* variable as the first boot option on the next boot, and only on the next boot. The boot manager removes the *BootNext* variable before transferring control to the *BootNext* boot option. If the boot from the *BootNext* boot option fails the boot sequence continues utilizing the BootOrder variable. If the boot from the *BootNext* boot option succeeds by returning **EFI_SUCCESS** the boot manager will not continue to boot utilizing the *BootOrder* variable.

The boot manager must call **LoadImage()** which supports at least **SIMPLE_FILE_PROTOCOL** and **LOAD_FILE_PROTOCOL** for resolving load options. If **LoadImage()** succeeds, the boot manager must enable the watchdog timer for 5 minutes by using the **SetWatchdogTimer()** boot service prior to calling **StartImage()**. If a boot option returns control to the boot manager, the boot manager must disable the watchdog timer with an additional call to the **SetWatchdogTimer()** boot service.

If the boot image is not loaded via **LoadImage()** the boot manager is required to check for a default application to boot. Searching for a default application to boot happens on both removable and fixed media types. This search occurs when the device path of the boot image listed in any boot option points directly to a **SIMPLE_FILE_SYSTEM** device and does not specify the exact file to load. The file discovery method is explained in "Boot Option Variables Default Behavior" starting on page 2-7 of this chapter. The default media boot case of a protocol other than **SIMPLE_FILE_SYSTEM** is handled by the **LOAD_FILE_PROTOCOL** for the target device path and does not need to be handled by the boot manager.

The boot manager must also support booting from a short-form device path that starts with the first element being a hard drive media device path (see Table 8-24, "Hard Drive Media Device Path" in Chapter 8). The boot manager must use the GUID or signature and partition number in the hard drive device path to match it to a device in the system. If the drive supports the GPT partitioning scheme the GUID in the hard drive media device path is compared with the *UniquePartitionGuid* field of the GUID Partition Entry (see Table 11-2 in Chapter 11). If the drive supports the PC-AT MBR scheme the signature in the hard drive media device path is compared with the *UniqueMBRSignature* in the Legacy Master Boot Record (see Table 11-5 in Chapter 11). If a signature match is made, then the partition number must also be matched. The hard drive device path can be appended to the matching hardware device path and normal boot behavior can then be used. If more than one device matches the hard drive device path, the boot manager will pick one arbitrarily. Thus the operating system must ensure the uniqueness of the signatures on hard drives to guarantee deterministic boot behavior.

Each load option variable contains an **EFI_LOAD_OPTION** descriptor that is a byte packed buffer of variable length fields. Since some of the fields are variable length, an **EFI_LOAD_OPTION** cannot be described as a standard C data structure. Instead, the fields are listed below in the order that they appear in an **EFI_LOAD_OPTION** descriptor:

Descriptor

```
UINT32            Attributes;
UINT16            FilePathListLength;
CHAR16            Description[];
EFI_DEVICE_PATH   FilePathList[];
UINT8             OptionalData[];
```

Parameters

Attributes	The attributes for this load option entry. All unused bits must be zero and are reserved by the EFI specification for future growth. See "Related Definitions."
FilePathListLength	Length in bytes of the *FilePathList*. *OptionalData* starts at offset **sizeof(UINT32)** + **sizeof(UINT16)** + **StrSize(***Description***)** + *FilePathListLength* of the **EFI_LOAD_OPTION** descriptor.
Description	The user readable description for the load option. This field ends with a Null Unicode character.

FilePathList	A packed array of EFI device paths. The first element of the array is an EFI device path that describes the device and location of the Image for this load option. The *FilePathList[0]* is specific to the device type. Other device paths may optionally exist in the *FilePathList*, but their usage is OSV specific. Each element in the array is variable length, and ends at the device path end structure. Because the size of *Description* is arbitrary, this data structure is not guaranteed to be aligned on a natural boundary. This data structure may have to be copied to an aligned natural boundary before it is used.
OptionalData	The remaining bytes in the load option descriptor are a binary data buffer that is passed to the loaded image. If the field is zero bytes long, a Null pointer is passed to the loaded image. The number of bytes in *OptionalData* can be computed by subtracting the starting offset of *OptionalData* from total size in bytes of the **EFI_LOAD_OPTION**.

Related Definitions

```
//*******************************************************
// Attributes
//*******************************************************
#define LOAD_OPTION_ACTIVE              0x00000001
#define LOAD_OPTION_FORCE_RECONNECT     0x00000002
```

Description

Calling **SetVariable()** creates a load option. The size of the load option is the same as the size of the *DataSize* argument to the **SetVariable()** call that created the variable. When creating a new load option, all undefined attribute bits must be written as zero. When updating a load option, all undefined attribute bits must be preserved. If a load option is not marked as **LOAD_OPTION_ACTIVE,** the boot manager will not automatically load the option. This provides an easy way to disable or enable load options without needing to delete and re-add them. If any *Driver####* load option is marked as **LOAD_OPTION_FORCE_RECONNECT**, then all of the EFI drivers in the system will be disconnected and reconnected after the last *Driver####* load option is processed. This allows an EFI driver loaded with a *Driver####* load option to override an EFI driver that was loaded prior to the execution of the EFI Boot Manager.

3.2 Globally-Defined Variables

This section defines a set of variables that have architecturally defined meanings. In addition to the defined data content, each such variable has an architecturally defined attribute that indicates when the data variable may be accessed. The variables with an attribute of NV are nonvolatile. This means that their values are persistent across resets and power cycles. The value of any environment variable that does not have this attribute will be lost when power is removed from the system and the state of firmware reserved memory is not otherwise preserved. The variables with an attribute of BS are only available before **ExitBootServices()** is called. This means that these environment variables can only be retrieved or modified in the preboot environment. They are not visible to an operating system. Environment variables with an attribute of RT are available before and after **ExitBootServices()** is called. Environment variables of this type can be retrieved and modified in the preboot environment, and from an operating system. All architecturally defined variables use the **EFI_GLOBAL_VARIABLE** *VendorGuid*:

```
#define EFI_GLOBAL_VARIABLE       \
     {8BE4DF61-93CA-11d2-AA0D-00E098032B8C}
```

To prevent name collisions with possible future globally defined variables, other internal firmware data variables that are not defined here must be saved with a unique *VendorGuid* other than **EFI_GLOBAL_VARIABLE**. Table 3-1 lists the global variables.

Table 3-1 Global Variables

Variable Name	Attribute	Description
LangCodes	BS, RT	The language codes that the firmware supports.
Lang	NV, BS, RT	The language code that the system is configured for.
Timeout	NV, BS, RT	The firmware's boot managers timeout, in seconds, before initiating the default boot selection.
ConIn	NV, BS, RT	The device path of the default input console.
ConOut	NV, BS, RT	The device path of the default output console.
ErrOut	NV, BS, RT	The device path of the default error output device.
ConInDev	BS, RT	The device path of all possible console input devices.
ConOutDev	BS, RT	The device path of all possible console output devices.
ErrOutDev	BS, RT	The device path of all possible error output devices.
Boot####	NV, BS, RT	A boot load option. #### is a printed hex value. No 0x or h is included in the hex value.
BootOrder	NV, BS, RT	The ordered boot option load list.
BootNext	NV, BS, RT	The boot option for the next boot only.
BootCurrent	BS, RT	The boot option that was selected for the current boot.
Driver####	NV, BS, RT	A driver load option. #### is a printed hex value.
DriverOrder	NV, BS, RT	The ordered driver load option list.

The *LangCodes* variable contains an array of 3-character (8-bit ASCII characters) ISO-639-2 language codes that the firmware can support. At initialization time the firmware computes the supported languages and creates this data variable. Since the firmware creates this value on each initialization, its contents are not stored in nonvolatile memory. This value is considered read-only.

The *Lang* variable contains the 3-character (8-bit ASCII characters) ISO-639-2 language code that the machine has been configured for. This value may be changed to any value supported by *LangCodes*; however, the change does not take effect until the next boot. If the language code is set to an unsupported value, the firmware will choose a supported default at initialization and set *Lang* to a supported value.

The *Timeout* variable contains a binary **UINT16** that supplies the number of seconds that the firmware will wait before initiating the original default boot selection. A value of 0 indicates that the default boot selection is to be initiated immediately on boot. If the value is not present, or contains the value of 0xFFFF then firmware will wait for user input before booting. This means the default boot selection is not automatically started by the firmware.

The *ConIn*, *ConOut*, and *ErrOut* variables each contain an **EFI_DEVICE_PATH** descriptor that defines the default device to use on boot. Changes to these values do not take effect until the next boot. If the firmware cannot resolve the device path, it is allowed to automatically replace the value(s) as needed to provide a console for the system.

The *ConInDev*, *ConOutDev*, and *ErrOutDev* variables each contain an **EFI_DEVICE_PATH** descriptor that defines all the possible default devices to use on boot. These variables are volatile, and are set dynamically on every boot. *ConIn*, *ConOut*, and *ErrOut* are always proper subsets of *ConInDev*, *ConOutDev*, and *ErrOutDev*.

Each *Boot####* variable contains an **EFI_LOAD_OPTION**. Each *Boot####* variable is the name "Boot" appended with a unique four digit hexadecimal number. For example, Boot0001, Boot0002, Boot0A02, etc.

The *BootOrder* variable contains an array of **UINT16**'s that make up an ordered list of the *Boot####* options. The first element in the array is the value for the first logical boot option, the second element is the value for the second logical boot option, etc. The *BootOrder* order list is used by the firmware's boot manager as the default boot order.

The *BootNext* variable is a single **UINT16** that defines the *Boot####* option that is to be tried first on the next boot. After the *BootNext* boot option is tried the normal *BootOrder* list is used. To prevent loops, the boot manager deletes this variable before transferring control to the preselected boot option.

The *BootCurrent* variable is a single **UINT16** that defines the *Boot####* option that was selected on the current boot.

Each *Driver####* variable contains an **EFI_LOAD_OPTION**. Each load option variable is appended with a unique number, for example Driver0001, Driver0002, etc.

The *DriverOrder* variable contains an array of **UINT16**'s that make up an ordered list of the *Driver####* variable. The first element in the array is the value for the first logical driver load option, the second element is the value for the second logical driver load option, etc. The *DriverOrder* list is used by the firmware's boot manager as the default load order for EFI drivers that it should explicitly load.

3.3 Boot Option Variables Default Behavior

The default state of globally-defined variables is firmware vendor specific. However the boot options require a standard default behavior in the exceptional case that valid boot options are not present on a platform. The default behavior must be invoked any time the *`BootOrder`* variable does not exist or only points to nonexistent boot options.

If no valid boot options exist, the boot manager will enumerate all removable EFI media devices followed by all fixed EFI media devices. The order within each group is undefined. These new default boot options are not saved to non volatile storage. The boot manger will then attempt to boot from each boot option. If the device supports the **`SIMPLE_FILE_SYSTEM`** protocol then the removable media boot behavior (see section 3.4.1.1) is executed. Otherwise the firmware will attempt to boot the device via the **`LOAD_FILE`** protocol .

It is expected that this default boot will load an operating system or a maintenance utility. If this is an operating system setup program it is then responsible for setting the requisite environment variables for subsequent boots. The platform firmware may also decide to recover or set to a known set of boot options.

3.4 Boot Mechanisms

EFI can boot from a device using the **`SIMPLE_FILE_SYSTEM`** protocol or the **`LOAD_FILE`** protocol. A device that supports the **`SIMPLE_FILE_SYSTEM`** protocol must materialize a file system protocol for that device to be bootable. If a device does not wish to support a complete file system it may produce a **`LOAD_FILE`** protocol which allows it to materialize an image directly. The Boot Manager will attempt to boot using the **`SIMPLE_FILE_SYSTEM`** protocol first. If that fails, then the **`LOAD_FILE`** protocol will be used.

3.4.1 Boot via Simple File Protocol

When booting via the **`SIMPLE_FILE_SYSTEM`** protocol, the *`FilePath`* will start with a device path that points to the device that "speaks" the **`SIMPLE_FILE_SYSTEM`** protocol. The next part of the *`FilePath`* will point to the file name, including sub directories that contain the bootable image. If the file name is a null device path, the file name must be discovered on the media using the rules defined for removable media devices with ambiguous file names (see section 3.4.1.1 below).

The format of the file system specified by EFI is contained in Chapter 11. While the firmware must produce a **`SIMPLE_FILE_SYSTEM`** protocol that understands the EFI file system, any file system can be abstracted with the **`SIMPLE_FILE_SYSTEM`** protocol interface.

3.4.1.1 Removable Media Boot Behavior

On a removable media device it is not possible for the *`FilePath`* to contain a file name, including sub directories. The *`FilePath`* is stored in non volatile memory in the platform and cannot possibly be kept in sync with a media that can change at any time. A *`FilePath`* for a removable media device will point to a device that "speaks" the **`SIMPLE_FILE_SYSTEM`** protocol. The *`FilePath`* will not contain a file name or sub directories.

The system firmware will attempt to boot from a removable media *FilePath* by adding a default file name in the form \EFI\BOOT\BOOT{machine type short-name}.EFI. Where machine type short-name defines a PE32+ image format architecture. Each file only contains one EFI image type, and a system may support booting from one or more images types. Table 3-2 lists the EFI image types.

Table 3-2 EFI Image Types

Architecture	File name convention	PE Executable machine type *
IA-32	BOOTIA32.EFI	0x14c
Itanium architecture	BOOTIA64.EFI	0x200

Note: * The PE Executable machine type is contained in the machine field of the COFF file header as defined in the Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0

A media may support multiple architectures by simply having a \EFI\BOOT\BOOT{machine type short-name}.EFI file of each possible machine type.

3.4.2 Boot via LOAD_FILE Protocol

When booting via the **LOAD_FILE** protocol, the *FilePath* is a device path that points to a device that "speaks" the **LOAD_FILE** protocol. The image is loaded directly from the device that supports the **LOAD_FILE** protocol. The remainder of the *FilePath* will contain information that is specific to the device. EFI firmware passes this device-specific data to the loaded image, but does not use it to load the image. If the remainder of the *FilePath* is a null device path it is the loaded image's responsibility to implement a policy to find the correct boot device.

The **LOAD_FILE** protocol is used for devices that do not directly support file systems. Network devices commonly boot in this model where the image is materialized without the need of a file system.

3.4.2.1 Network Booting

Network booting is described by the *Preboot eXecution Environment (PXE) BIOS Support Specification* that is part of the *Wired for Management Baseline specification*. PXE specifies UDP, DHCP, and TFTP network protocols that a booting platform can use to interact with an intelligent system load server. EFI defines special interfaces that are used to implement PXE. These interfaces are contained in the PXE_BASE_CODE protocol (Chapter 15).

3.4.2.2 Future Boot Media

Since EFI defines an abstraction between the platform and the OS and its loader it should be possible to add new types of boot media as technology evolves. The OS loader will not necessarily have to change to support new types of boot. The implementation of the EFI platform services may change, but the interface will remain constant. The OS will require a driver to support the new type of boot media so that it can make the transition from EFI boot services to OS control of the boot media.

4
EFI System Table

This chapter describes the entry point to an EFI image and the parameters that are passed to that entry point. There are three types of EFI images that can be loaded and executed by EFI firmware. These are EFI Applications, EFI OS Loaders, and EFI Drivers. There are no differences in the entry point for these three image types.

4.1 EFI Image Entry Point

The most significant parameter that is passed to an EFI image is a pointer to the EFI System Table. This pointer is **EFI_IMAGE_ENTRY_POINT** (see definition immediately below), the main entry point for an EFI Image. The EFI System Table contains pointers to the active console devices, a pointer to the EFI Boot Services Table, a pointer to the EFI Runtime Services Table, and a pointer to the list of system configuration tables such as ACPI, SMBIOS, and the SAL System Table. This chapter describes the EFI System Table in detail.

EFI_IMAGE_ENTRY_POINT

Summary

This is the main entry point for an EFI Image. This entry point is the same for EFI Applications, EFI OS Loaders, and EFI Drivers including both device drivers and bus drivers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IMAGE_ENTRY_POINT) (
  IN  EFI_HANDLE          ImageHandle,
  IN  EFI_SYSTEM_TABLE    *SystemTable
  );
```

Parameters

ImageHandle	The firmware allocated handle for the EFI image.
SystemTable	A pointer to the EFI System Table.

Description

This function is the entry point to an EFI image. An EFI image is loaded and relocated in system memory by the EFI Boot Service **LoadImage()**. An EFI image is invoked through the EFI Boot Service **StartImage()**.

The first argument is the image's image handle. The second argument is a pointer to the image's system table. The system table contains the standard output and input handles, plus pointers to the **EFI_BOOT_SERVICES** and **EFI_RUNTIME_SERVICES** tables. The service tables contain the entry points in the firmware for accessing the core EFI system functionality. The handles in the system table are used to obtain basic access to the console. In addition, the EFI system table contains pointers to other standard tables that a loaded image may use if the associated pointers are initialized to nonzero values. Examples of such tables are ACPI, SMBIOS, SAL System Table, etc.

The *ImageHandle* is a firmware-allocated handle that is used to identify the image on various functions. The handle also supports one or more protocols that the image can use. All images support the **EFI_LOADED_IMAGE** protocol that returns the source location of the image, the memory location of the image, the load options for the image, etc. The exact **EFI_LOADED_IMAGE** structure is defined in Chapter 7.

If the EFI image is an EFI Application, then the EFI Application executes and either returns or calls the EFI Boot Services **Exit()**. An EFI Application is always unloaded from memory when it exits, and its return status is returned to the component that started the EFI Application.

If the EFI image is an EFI OS Loader, then the EFI OS Loader executes and either returns, calls the EFI Boot Service **Exit(),** or calls the EFI Boot Service **ExitBootServices()**. If the EFI OS Loader returns or calls **Exit()**, then the load of the OS has failed, and the EFI OS Loader is unloaded from memory and control is returned to the component that attempted to boot the EFI OS Loader. If **ExitBootServices()** is called, then the OS Loader has taken control of the platform, and EFI will not regain control of the system until the platform is reset. One method of resetting the platform is through the EFI Runtime Service **ResetSystem()**.

If the EFI image is an EFI Driver, then the EFI Driver executes and either returns or calls the EFI Boot Service **Exit()**. If an EFI driver returns an error, then the driver is unloaded from memory. If the EFI driver returns **EFI_SUCCESS**, then it stays resident in memory. If the EFI Driver does not follow the EFI Driver Model, then it performs any required initialization and installs its protocol services before returning. If the EFI driver does follow the EFI Driver Model, then the entry point is not allowed to touch any device hardware. Instead, the entry point is required to create and install the **EFI_DRIVER_BINDING_PROTOCOL** (Chapter 9) on the *ImageHandle* of the EFI Driver. If this process is completed, then **EFI_SUCCESS** is returned. If the resources are not available to complete the driver initialization, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The driver was initialized.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

4.2 EFI Table Header

The data type **`EFI_TABLE_HEADER`** is the data structure that precedes all of the standard EFI table types. It includes a signature that is unique for each table type, a revision of the table that may be updated as extensions are added to the EFI table types, and a 32-bit CRC so a consumer of an EFI table type can validate the contents of the EFI table.

EFI_TABLE_HEADER

Summary

Data structure that precedes all of the standard EFI table types.

Related Definitions

```
typedef struct {
  UINT64     Signature;
  UINT32     Revision;
  UINT32     HeaderSize;
  UINT32     CRC32;
  UINT32     Reserved;
} EFI_TABLE_HEADER;
```

Parameters

`Signature`	A 64-bit signature that identifies the type of table that follows. Unique signatures have been generated for the EFI System Table, the EFI Boot Services Table, and the EFI Runtime Services Table.
`Revision`	The revision of the EFI Specification to which this table conforms. The upper 16 bits of this field contain the major revision value, and the lower 16 bits contain the minor revision value. The minor revision values are limited to the range of 00..99.
`HeaderSize`	The size, in bytes, of the entire table including the **`EFI_TABLE_HEADER`**.
`CRC32`	The 32-bit CRC for the entire table. This value is computed by setting this field to 0, and computing the 32-bit CRC for *`HeaderSize`* bytes.
`Reserved`	Reserved field that must be set to 0.

NOTE

The size of the EFI system table, runtime services table, and boot services table may increase over time. It is very important to always use the `HeaderSize` field of the **`EFI_TABLE_HEADER`** *to determine the size of these tables.*

4.3 EFI System Table

The EFI System Table contains pointers to the runtime and boot services tables. The definition for this table is shown in the following code fragments. Except for the table header, all elements in the service tables are prototypes of function pointers to functions as defined in Chapters 5 and 6. Prior to a call to **ExitBootServices()**, all of the fields of the EFI System Table are valid. After an operating system has taken control of the platform with a call to **ExitBootServices()**, only the *Hdr*, *FirmwareVendor*, *FirmwareRevision*, *RuntimeServices*, *NumberOfTableEntries*, and *ConfigurationTable* fields are valid.

EFI_SYSTEM_TABLE

Summary

Contains pointers to the runtime and boot services tables.

Related Definitions

```
#define EFI_SYSTEM_TABLE_SIGNATURE      0x5453595320494249
#define EFI_SYSTEM_TABLE_REVISION       ((1<<16) | (10))
#define EFI_1_10_SYSTEM_TABLE_REVISION  ((1<<16) | (10))
#define EFI_1_02_SYSTEM_TABLE_REVISION  ((1<<16) | (02))

typedef struct {
  EFI_TABLE_HEADER              Hdr;
  CHAR16                        *FirmwareVendor;
  UINT32                        FirmwareRevision;
  EFI_HANDLE                    ConsoleInHandle;
  SIMPLE_INPUT_INTERFACE        *ConIn;
  EFI_HANDLE                    ConsoleOutHandle;
  SIMPLE_TEXT_OUTPUT_INTERFACE  *ConOut;
  EFI_HANDLE                    StandardErrorHandle;
  SIMPLE_TEXT_OUTPUT_INTERFACE  *StdErr;
  EFI_RUNTIME_SERVICES          *RuntimeServices;
  EFI_BOOT_SERVICES             *BootServices;
  UINTN                         NumberOfTableEntries;
  EFI_CONFIGURATION_TABLE       *ConfigurationTable;
} EFI_SYSTEM_TABLE;
```

Parameters

Hdr	The table header for the EFI System Table. This header contains the **EFI_SYSTEM_TABLE_SIGNATURE** and **EFI_SYSTEM_TABLE_REVISION** values along with the size of the **EFI_SYSTEM_TABLE** structure and a 32-bit CRC to verify that the contents of the EFI System Table are valid.
FirmwareVendor	A pointer to a null terminated Unicode string that identifies the vendor that produces the system firmware for the platform.
FirmwareRevision	A firmware vendor specific value that identifies the revision of the system firmware for the platform.
ConsoleInHandle	The handle for the active console input device. This handle must support the **SIMPLE_INPUT_PROTOCOL**.
ConIn	A pointer to the **SIMPLE_INPUT_PROTOCOL** interface that is associated with *ConsoleInHandle*.
ConsoleOutHandle	The handle for the active console output device. This handle must support the **SIMPLE_TEXT_OUTPUT_PROTOCOL**.
ConOut	A pointer to the **SIMPLE_TEXT_OUTPUT_PROTOCOL** interface that is associated with *ConsoleOutHandle*.
StandardErrorHandle	The handle for the active standard error console device. This handle must support the **SIMPLE_TEXT_OUTPUT_PROTOCOL**.
StdErr	A pointer to the **SIMPLE_TEXT_OUTPUT_PROTOCOL** interface that is associated with *StandardErrorHandle*.
RuntimeServices	A pointer to the EFI Runtime Services Table. See Section 4.5.
BootServices	A pointer to the EFI Boot Services Table. See Section 4.4.
NumberOfTableEntries	The number of system configuration tables in the buffer *ConfigurationTable*.
ConfigurationTable	A pointer to the system configuration tables. The number of entries in the table is *NumberOfTableEntries*.

4.4 EFI Boot Services Table

The EFI Boot Services Table contains a table header and pointers to all of the boot services. The definition for this table is shown in the following code fragments. Except for the table header, all elements in the EFI Boot Services Tables are prototypes of function pointers to functions as defined in Chapters 5. The function pointers in this table are not valid after the operating system has taken control of the platform with a call to **ExitBootServices()**.

EFI_BOOT_SERVICES

Summary

Contains a table header and pointers to all of the boot services.

Related Definitions

```
#define EFI_BOOT_SERVICES_SIGNATURE     0x56524553544f4f42
#define EFI_BOOT_SERVICES_REVISION      ((1<<16) | (10))

typedef struct {
  EFI_TABLE_HEADER                 Hdr;

  //
  // Task Priority Services
  //
  EFI_RAISE_TPL                    RaiseTPL;
  EFI_RESTORE_TPL                  RestoreTPL;

  //
  // Memory Services
  //
  EFI_ALLOCATE_PAGES               AllocatePages;
  EFI_FREE_PAGES                   FreePages;
  EFI_GET_MEMORY_MAP               GetMemoryMap;
  EFI_ALLOCATE_POOL                AllocatePool;
  EFI_FREE_POOL                    FreePool;

  //
  // Event & Timer Services
  //
  EFI_CREATE_EVENT                 CreateEvent;
  EFI_SET_TIMER                    SetTimer;
  EFI_WAIT_FOR_EVENT               WaitForEvent;
  EFI_SIGNAL_EVENT                 SignalEvent;
  EFI_CLOSE_EVENT                  CloseEvent;
  EFI_CHECK_EVENT                  CheckEvent;
```

```
//
// Protocol Handler Services
//
EFI_INSTALL_PROTOCOL_INTERFACE    InstallProtocolInterface;
EFI_REINSTALL_PROTOCOL_INTERFACE  ReinstallProtocolInterface;
EFI_UNINSTALL_PROTOCOL_INTERFACE  UninstallProtocolInterface;
EFI_HANDLE_PROTOCOL               HandleProtocol;
VOID                              *Reserved;
EFI_REGISTER_PROTOCOL_NOTIFY      RegisterProtocolNotify;
EFI_LOCATE_HANDLE                 LocateHandle;
EFI_LOCATE_DEVICE_PATH            LocateDevicePath;
EFI_INSTALL_CONFIGURATION_TABLE   InstallConfigurationTable;

//
// Image Services
//
EFI_IMAGE_LOAD                    LoadImage;
EFI_IMAGE_START                   StartImage;
EFI_EXIT                          Exit;
EFI_IMAGE_UNLOAD                  UnloadImage;
EFI_EXIT_BOOT_SERVICES            ExitBootServices;

//
// Miscellaneous Services
//
EFI_GET_NEXT_MONOTONIC_COUNT      GetNextMonotonicCount;
EFI_STALL                         Stall;
EFI_SET_WATCHDOG_TIMER            SetWatchdogTimer;

//
// DriverSupport Services
//
EFI_CONNECT_CONTROLLER            ConnectController;
EFI_DISCONNECT_CONTROLLER         DisconnectController;

//
// Open and Close Protocol Services
//
EFI_OPEN_PROTOCOL                 OpenProtocol;
EFI_CLOSE_PROTOCOL                CloseProtocol;
EFI_OPEN_PROTOCOL_INFORMATION     OpenProtocolInformation;

//
// Library Services
//
EFI_PROTOCOLS_PER_HANDLE          ProtocolsPerHandle;
EFI_LOCATE_HANDLE_BUFFER          LocateHandleBuffer;
EFI_LOCATE_PROTOCOL               LocateProtocol;
```

```
    EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES
                                      InstallMultipleProtocolInterfaces;
    EFI_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES
                                      IninstallMultipleProtocolInterfaces;

    //
    // 32-bit CRC Services
    //
    EFI_CALCULATE_CRC32                  CalculateCrc32;

    //
    // Memory Utility Services
    //
    EFI_COPY_MEM                         CopyMem;
    EFI_SET_MEM                          SetMem;

  } EFI_BOOT_SERVICES;
```

Parameters

Hdr	The table header for the EFI Boot Services Table. This header contains the **EFI_BOOT_SERVICES_ SIGNATURE** and **EFI_BOOT_SERVICES_ REVISION** values along with the size of the **EFI_BOOT_SERVICES_TABLE** structure and a 32-bit CRC to verify that the contents of the EFI Boot Services Table are valid.
RaiseTPL	Raises the task priority level.
RestoreTPL	Restores/lowers the task priority level.
AllocatePages	Allocates pages of a particular type.
FreePages	Frees allocated pages.
GetMemoryMap	Returns the current boot services memory map and memory map key.
AllocatePool	Allocates a pool of a particular type.
FreePool	Frees allocated pool.
CreateEvent	Creates a general-purpose event structure.
SetTimer	Sets an event to be signaled at a particular time.
WaitForEvent	Stops execution until an event is signaled.
SignalEvent	Signals an event.
CloseEvent	Closes and frees an event structure.

CheckEvent	Checks whether an event is in the signaled state.
InstallProtocolInterface	Installs a protocol interface on a device handle.
ReinstallProtocolInterface	Reinstalls a protocol interface on a device handle.
UninstallProtocolInterface	Removes a protocol interface from a device handle.
HandleProtocol	Queries a handle to determine if it supports a specified protocol.
Reserved	Reserved. Must be **NULL**.
RegisterProtocolNotify	Registers an event that is to be signaled whenever an interface is installed for a specified protocol.
LocateHandle	Returns an array of handles that support a specified protocol.
LocateDevicePath	Locates all devices on a device path that support a specified protocol and returns the handle to the device that is closest to the path.
InstallConfigurationTable	Adds, updates, or removes a configuration table from the EFI System Table.
LoadImage	Loads an EFI image into memory.
StartImage	Transfers control to a loaded image's entry point.
Exit	Exits the image's entry point.
UnloadImage	Unloads an image.
ExitBootServices	Terminates boot services.
GetNextMonotonicCount	Returns a monotonically increasing count for the platform.
Stall	Stalls the processor.
SetWatchdogTimer	Resets and sets a watchdog timer used during boot services time.
ConnectController	Uses a set of precedence rules to find the best set of drivers to manage a controller.
DisconnectController	Informs a set of drivers to stop managing a controller.
OpenProtocol	Adds elements to the list of agents consuming a protocol interface.
CloseProtocol	Removes elements from the list of agents consuming a protocol interface.

`OpenProtocolInformation`	Retrieve the list of agents that are currently consuming a protocol interface.
`ProtocolsPerHandle`	Retrieves the list of protocols installed on a handle. The return buffer is automatically allocated.
`LocateHandleBuffer`	Retrieves the list of handles from the handle database that meet the search criteria. The return buffer is automatically allocated.
`LocateProtocol`	Finds the first handle in the handle database the supports the requested protocol.
`InstallMultipleProtocolInterfaces`	Installs one or more protocol interfaces onto a handle.
`UninstallMultipleProtocolInterfaces`	Uninstalls one or more protocol interfaces from a handle.
`CalculateCrc32`	Computes and returns a 32-bit CRC for a data buffer.
`CopyMem`	Copies the contents of one buffer to another buffer.
`SetMem`	Fills a buffer with a specified value.

4.5 EFI Runtime Services Table

The EFI Runtime Services Table contains a table header and pointers to all of the runtime services. The definition for this table is shown in the following code fragments. Except for the table header, all elements in the EFI Runtime Services Tables are prototypes of function pointers to functions as defined in Chapters 6. Unlike the EFI Boot Services Table, this table, and the function pointers it contains are valid after the operating system has taken control of the platform with a call to `ExitBootServices()`. If a call to `SetVirtualAddressMap()` is made by the OS, then the function pointers in this table are fixed up to point to the new virtually mapped entry points.

EFI_RUNTIME_SERVICES

Summary

Contains a table header and pointers to all of the runtime services.

Related Definitions

```
#define EFI_RUNTIME_SERVICES_SIGNATURE  0x56524553544e5552
#define EFI_RUNTIME_SERVICES_REVISION   ((1<<16) | (10))

typedef struct {
  EFI_TABLE_HEADER                Hdr;

  //
  // Time Services
  //
  EFI_GET_TIME                    GetTime;
  EFI_SET_TIME                    SetTime;
  EFI_GET_WAKEUP_TIME             GetWakeupTime;
  EFI_SET_WAKEUP_TIME             SetWakeupTime;

  //
  // Virtual Memory Services
  //
  EFI_SET_VIRTUAL_ADDRESS_MAP     SetVirtualAddressMap;
  EFI_CONVERT_POINTER             ConvertPointer;

  //
  // Variable Services
  //
  EFI_GET_VARIABLE                GetVariable;
  EFI_GET_NEXT_VARIABLE_NAME      GetNextVariableName;
  EFI_SET_VARIABLE                SetVariable;
  //
  // Miscellaneous Services
  //
  EFI_GET_NEXT_HIGH_MONO_COUNT     GetNextHighMonotonicCount;
  EFI_RESET_SYSTEM                 ResetSystem;
} EFI_RUNTIME_SERVICES;
```

Parameters

Hdr	The table header for the EFI Runtime Services Table. This header contains the **EFI_RUNTIME_SERVICES_SIGNATURE** and **EFI_RUNTIME_SERVICES_REVISION** values along with the size of the **EFI_RUNTIME_SERVICES_TABLE** structure and a 32-bit CRC to verify that the contents of the EFI Runtime Services Table are valid.
GetTime	Returns the current time and date, and the time-keeping capabilities of the platform.
SetTime	Sets the current local time and date information.
GetWakeupTime	Returns the current wakeup alarm clock setting.
SetWakeupTime	Sets the system wakeup alarm clock time.
SetVirtualAddressMap	Used by an OS loader to convert from physical addressing to virtual addressing.
ConvertPointer	Used by EFI components to convert internal pointers when switching to virtual addressing.
GetVariable	Returns the value of a variable.
GetNextVariableName	Enumerates the current variable names.
SetVariable	Sets the value of a variable.
GetNextHighMonotonicCount	Returns the next high 32 bits of the platform's monotonic counter.
ResetSystem	Resets the entire platform.

4.6 EFI Configuration Table

The EFI Configuration Table is the *ConfigurationTable* field in the EFI System Table. This table contains a set of GUID/pointer pairs. Each element of this table is described by the **EFI_CONFIGURATION_TABLE** structure below. The number of types of configuration tables is expected to grow over time. This is why a GUID is used to identify the configuration table type. The EFI Configuration Table may contain at most once instance of each table type. The list of current configuration table types is also listed below.

EFI_CONFIGURATION_TABLE

Summary

Contains a set of GUID/pointer pairs comprised of the *ConfigurationTable* field in the EFI System Table.

Related Definitions

```
typedef struct{
  EFI_GUID                          VendorGuid;
  VOID                              *VendorTable;
} EFI_CONFIGURATION_TABLE;
```

Parameters

VendorGuid	The 128-bit GUID value that uniquely identifies the system configuration table.
VendorTable	A pointer to the table associated with *VendorGuid*.

```
#define ACPI_20_TABLE_GUID  \
  {0x8868e871,0xe4f1,0x11d3,0xbc,0x22,0x0,0x80,0xc7,0x3c,0x88,0x81}

#define ACPI_TABLE_GUID \
  {0xeb9d2d30,0x2d88,0x11d3,0x9a,0x16,0x0,0x90,0x27,0x3f,0xc1,0x4d}

#define SAL_SYSTEM_TABLE_GUID \
  {0xeb9d2d32,0x2d88,0x11d3,0x9a,0x16,0x0,0x90,0x27,0x3f,0xc1,0x4d}

#define SMBIOS_TABLE_GUID \
  {0xeb9d2d31,0x2d88,0x11d3,0x9a,0x16,0x0,0x90,0x27,0x3f,0xc1,0x4d}

#define MPS_TABLE_GUID \
  {0xeb9d2d2f,0x2d88,0x11d3,0x9a,0x16,0x0,0x90,0x27,0x3f,0xc1,0x4d}
```

4.7 EFI Image Entry Point Examples

The examples in the following sections show how the various table examples are presented in the EFI environment.

4.7.1 EFI Image Entry Point Examples

The following example shows the EFI image entry point for an EFI Application. This application makes use of the EFI System Table, the EFI Boot Services Table, and the EFI Runtime Services Table.

```
EFI_SYSTEM_TABLE                    *gST;
EFI_BOOT_SERVICES_TABLE             *gBS;
EFI_RUNTIME_SERVICES_TABLE          *gRT;

EfiApplicationEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS  Status;
  EFI_TIME    *Time;

  gST = SystemTable;
  gBS = gST->BootServices;
  gRT = gST->RuntimeServices;

  //
  // Use EFI System Table to print "Hello World" to the active console output
  // device.
  //
  Status = gST->ConOut->OutputString (gST->ConOut, L"Hello World\n\r");
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Use EFI Boot Services Table to allocate a buffer to store the current time
  // and date.
  //
  Status = gBS->AllocatePool (
                  EfiBootServicesData,
                  sizeof (EFI_TIME),
                  (VOID **)&Time
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }
```

```
    //
    // Use the EFI Runtime Services Table to get the current time and date.
    //
    Status = gRT->GetTime (&Time, NULL)
    if (EFI_ERROR (Status)) {
      return Status;
    }

    return Status;
  }
```

The following example shows the EFI image entry point for an EFI Driver that does not follow the *EFI Driver Model*. Since this driver returns **EFI_SUCCESS**, it will stay resident in memory after it exits.

```
EFI_SYSTEM_TABLE                    *gST;
EFI_BOOT_SERVICES_TABLE             *gBS;
EFI_RUNTIME_SERVICES_TABLE          *gRT;

EfiDriverEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  gST = SystemTable;
  gBS = gST->BootServices;
  gRT = gST->RuntimeServices;

  //
  // Implement driver initialization here.
  //

  return EFI_SUCCESS;
}
```

The following example shows the EFI image entry point for an EFI Driver that also does not follow the *EFI Driver Model*. Since this driver returns **EFI_DEVICE_ERROR**, it will not stay resident in memory after it exits.

```
EFI_SYSTEM_TABLE                    *gST;
EFI_BOOT_SERVICES_TABLE             *gBS;
EFI_RUNTIME_SERVICES_TABLE          *gRT;

EfiDriverEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  gST = SystemTable;
  gBS = gST->BootServices;
  gRT = gST->RuntimeServices;
```

```
    //
    // Implement driver initialization here.
    //

    return EFI_DEVICE_ERROR;
  }
```

4.7.2 EFI Driver Model Example

The following is an *EFI Driver Model* example that shows the driver initialization routine for the ABC device controller that is on the XYZ bus. The **EFI_DRIVER_BINDING_PROTOCOL** is defined in Chapter 9. The function prototypes for the **AbcSupported()**, **AbcStart()**, and **AbcStop()** functions are defined in Section 9.1. This function saves the driver's image handle and a pointer to the EFI boot services table in global variables, so the other functions in the same driver can have access to these values. It then creates an instance of the **EFI_DRIVER_BINDING_PROTOCOL** and installs it onto the driver's image handle.

```
extern EFI_GUID                      gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES_TABLE              *gBS;
static EFI_DRIVER_BINDING_PROTOCOL   mAbcDriverBinding = {
  AbcSupported,
  AbcStart,
  AbcStop,
  1,
  NULL,
  NULL
};

AbcEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS  Status;

  gBS = SystemTable->BootServices;

  mAbcDriverBinding->ImageHandle         = ImageHandle;
  mAbcDriverBinding->DriverBindingHandle = ImageHandle;

  Status = gBS->InstallMultipleProtocolInterfaces(
                  &mAbcDriverBinding->DriverBindingHandle,
                  &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
                  NULL
                  );
  return Status;
}
```

4.7.3 EFI Driver Model Example (Unloadable)

The following is the same *EFI Driver Model* example as above, except it also includes the code required to allow the driver to be unloaded through the boot service **Unload()**. Any protocols installed or memory allocated in **AbcEntryPoint()** must be uninstalled or freed in the **AbcUnload()**.

```
extern EFI_GUID                     gEfiLoadedImageProtocolGuid;
extern EFI_GUID                     gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES_TABLE             *gBS;
static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBinding = {
  AbcSupported,
  AbcStart,
  AbcStop,
  1,
  NULL,
  NULL
};

EFI_STATUS
AbcUnload (
  IN EFI_HANDLE  ImageHandle
  );

AbcEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS                 Status;
  EFI_LOADED_IMAGE_PROTOCOL  *LoadedImage;

  gBS = SystemTable->BootServices;

  Status = gBS->OpenProtocol (
                ImageHandle,
                &gEfiLoadedImageProtocolGuid,
                &LoadedImage,
                ImageHandle,
                NULL,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  LoadedImage->Unload = AbcUnload;

  mAbcDriverBinding->ImageHandle         = ImageHandle;
  mAbcDriverBinding->DriverBindingHandle = ImageHandle;

  Status = gBS->InstallMultipleProtocolInterfaces(
                &mAbcDriverBinding->DriverBindingHandle,
                &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
                NULL
                );

  return Status;
}
```

```
EFI_STATUS
AbcUnload (
  IN EFI_HANDLE  ImageHandle
  )

{
  EFI_STATUS  Status;

  Status = gBS->UninstallMultipleProtocolInterfaces (
                  ImageHandle,
                  &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
                  NULL
                  );
  return Status;
}
```

4.7.4 EFI Driver Model Example (Multiple Instances)

The following is the same as the first *EFI Driver Model* example, except it produces three **EFI_DRIVER_BINDING_PROTOCOL** instances. The first one is installed onto the driver's image handle. The other two are installed onto newly created handles.

```
extern EFI_GUID                     gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES_TABLE             *gBS;

static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBindingA = {
  AbcSupportedA,
  AbcStartA,
  AbcStopA,
  1,
  NULL,
  NULL
};

static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBindingB = {
  AbcSupportedB,
  AbcStartB,
  AbcStopB,
  1,
  NULL,
  NULL
};

static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBindingC = {
  AbcSupportedC,
  AbcStartC,
  AbcStopC,
  1,
  NULL,
  NULL
};
```

```
AbcEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS  Status;

  gBS = SystemTable->BootServices;

  //
  // Install mAbcDriverBindingA onto ImageHandle
  //
  mAbcDriverBindingA->ImageHandle         = ImageHandle;
  mAbcDriverBindingA->DriverBindingHandle = ImageHandle;

  Status = gBS->InstallMultipleProtocolInterfaces(
                  &mAbcDriverBindingA->DriverBindingHandle,
                  &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingA,
                  NULL
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Install mAbcDriverBindingB onto a newly created handle
  //
  mAbcDriverBindingB->ImageHandle         = ImageHandle;
  mAbcDriverBindingB->DriverBindingHandle = NULL;

  Status = gBS->InstallMultipleProtocolInterfaces(
                  &mAbcDriverBindingB->DriverBindingHandle,
                  &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingB,
                  NULL
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Install mAbcDriverBindingC onto a newly created handle
  //
  mAbcDriverBindingC->ImageHandle         = ImageHandle;
  mAbcDriverBindingC->DriverBindingHandle = NULL;

  Status = gBS->InstallMultipleProtocolInterfaces(
                  &mAbcDriverBindingC->DriverBindingHandle,
                  &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingC,
                  NULL
                  );

  return Status;
}
```

5
Services — Boot Services

This chapter discusses the fundamental boot services that are present in an EFI-compliant system. The services are defined by interface functions that may be used by code running in the EFI environment. Such code may include protocols that manage device access or extend platform capability, as well as EFI applications running in the preboot environment and EFI OS loaders.

Two types of services apply in an EFI-compliant system:

- **Boot Services**. Functions that are available *before* a successful call to `ExitBootServices()`. These functions are described in this chapter.
- **Runtime Services**. Functions that are available *before and after* any call to `ExitBootServices()`. These functions are described in Chapter 6.

During boot, system resources are owned by the firmware and are controlled through boot services interface functions. These functions can be characterized as "global" or "handle-based." The term "global" simply means that a function accesses system services and is available on all platforms (since all platforms support all system services). The term "handle-based" means that the function accesses a specific device or device functionality and may not be available on some platforms (since some devices are not available on some platforms). Protocols are created dynamically. This chapter discusses the "global" functions and runtime functions; subsequent chapters discuss the "handle-based."

EFI applications (including OS loaders) must use boot services functions to access devices and allocate memory. On entry, an EFI Image is provided a pointer to an EFI system table which contains the Boot Services dispatch table and the default handles for accessing the console. All boot services functionality is available until an EFI OS loader loads enough of its own environment to take control of the system's continued operation and then terminates boot services with a call to `ExitBootServices()`.

In principle, the `ExitBootServices()` call is intended for use by the operating system to indicate that its loader is ready to assume control of the platform and all platform resource management. Thus boot services are available up to this point to assist the OS loader in preparing to boot the operating system. Once the OS loader takes control of the system and completes the operating system boot process, only runtime services may be called. Code other than the OS loader, however, may or may not choose to call `ExitBootServices()`. This choice may in part depend upon whether or not such code is designed to make continued use of EFI boot services or the boot services environment.

The rest of this chapter discusses individual functions. Global boot services functions fall into these categories:

- Event, Timer, and Task Priority Services (Section 5.1)
- Memory Allocation Services (Section 5.2)
- Protocol Handler Services (Section 5.3)
- Image Services (Section 5.4)
- Miscellaneous Services (Section 5.5)

5.1 Event, Timer, and Task Priority Services

The functions that make up the Event, Timer, and Task Priority Services are used during preboot to create, close, signal, and wait for events; to set timers; and to raise and restore task priority levels. See Table 5-1.

Table 5-1. Event, Timer, and Task Priority Functions

Name	Type	Description
CreateEvent	Boot	Creates a general-purpose event structure.
CloseEvent	Boot	Closes and frees an event structure.
SignalEvent	Boot	Signals an event.
WaitForEvent	Boot	Stops execution until an event is signaled.
CheckEvent	Boot	Checks whether an event is in the signaled state.
SetTimer	Boot	Sets an event to be signaled at a particular time.
RaiseTPL	Boot	Raises the task priority level.
RestoreTPL	Boot	Restores/lowers the task priority level.

Execution in the boot services environment occurs at different task priority levels, or TPLs. The boot services environment exposes only three of these levels to EFI applications and drivers:

- **TPL_APPLICATION,** the lowest priority level
- **TPL_CALLBACK**, an intermediate priority level
- **TPL_NOTIFY**, the highest priority level

Tasks that execute at a higher priority level may interrupt tasks that execute at a lower priority level. For example, tasks that run at the **TPL_NOTIFY** level may interrupt tasks that run at the **TPL_APPLICATION** or **TPL_CALLBACK** level. While **TPL_NOTIFY** is the highest level exposed to the boot services applications, the firmware may have higher task priority items it deals with. For example, the firmware may have to deal with tasks of higher priority like timer ticks and internal devices. Consequently, there is a fourth TPL, **TPL_HIGH_LEVEL**, designed for use exclusively by the firmware.

The intended usage of the priority levels is shown in Table 5-2 from the lowest level (**TPL_APPLICATION**) to the highest level (**TPL_HIGH_LEVEL**). As the level increases, the duration of the code and the amount of blocking allowed decrease. Execution generally occurs at the **TPL_APPLICATION** level. Execution occurs at other levels as a direct result of the triggering of an event notification function(this is typically caused by the signaling of an event). During timer interrupts, firmware signals timer events when an event's "trigger time" has expired. This allows event notification functions to interrupt lower priority code to check devices (for example). The notification function can signal other events as required. After all pending event notification functions execute, execution continues at the **TPL_APPLICATION** level.

Table 5-2. TPL Usage

Task Priority Level	Usage
TPL_APPLICATION	This is the lowest priority level. It is the level of execution which occurs when no event notifications are pending and which interacts with the user. User I/O (and blocking on User I/O) can be performed at this level. The boot manager executes at this level and passes control to other EFI applications at this level.
TPL_CALLBACK	Interrupts code executing below **TPL_CALLBACK** level. Long term operations (such as file system operations and disk I/O) can occur at this level.
TPL_NOTIFY	Interrupts code executing below **TPL_NOTIFY** level. Blocking is not allowed at this level. Code executes to completion and returns. If code requires more processing, it needs to signal an event to wait to reobtain control at whatever level it requires. This level is typically used to process low level IO to or from a device.
(Firmware Interrupts)	This level is internal to the firmware. It is the level at which internal interrupts occur. Code running at this level interrupts code running at the **TPL_NOTIFY** level (or lower levels). If the interrupt requires extended time to complete, firmware signals another event (or events) to perform the longer term operations so that other interrupts can occur.
TPL_HIGH_LEVEL	Interrupts code executing below **TPL_HIGH_LEVEL**. This is the highest priority level. It is not interruptable (interrupts are disabled) and is used sparingly by firmware to synchronize operations that need to be accessible from any priority level. For example, it must be possible to signal events while executing at any priority level. Therefore, firmware manipulates the internal event structure while at this priority level.

Executing code can temporarily raise its priority level by calling the **RaiseTPL()** function. Doing this masks event notifications from code running at equal or lower priority levels until the **RestoreTPL()** function is called to reduce the priority to a level below that of the pending event notifications. There are restrictions on the TPL levels at which many EFI service functions and protocol interface functions can execute. Table 5-3 summarizes the restrictions.

Table 5-3. TPL Restrictions

Name	Restriction	Task Priority Level
Memory Allocation Services	<=	TPL_NOTIFY
Variable Services	<=	TPL_CALLBACK
ExitBootServices()	=	TPL_APPLICATION
LoadImage()	<	TPL_CALLBACK
StartImage()	<	TPL_CALLBACK
UnloadImage()	<=	TPL_CALLBACK
Exit()	<=	TPL_CALLBACK
Time Services	<=	TPL_CALLBACK
WaitForEvent()	=	TPL_APPLICATION
SignalEvent()	<=	TPL_HIGH_LEVEL
Event Notification Levels	> <=	TPL_APPLICATION TPL_HIGH_LEVEL
Protocol Interface Functions	<=	TPL_NOTIIFY
Block I/O Protocol	<=	TPL_CALLBACK
Disk I/O Protocol	<=	TPL_CALLBACK
Simple File System Protocol	<=	TPL_CALLBACK
Simple Input Protocol	<=	TPL_APPLICATION
Simple Text Output Protocol	<=	TPL_NOTIFY
Serial I/O Protocol	<=	TPL_CALLBACK
PXE Base Code Protocol	<=	TPL_CALLBACK
Simple Network Protocol	<=	TPL_CALLBACK

CreateEvent()

Summary

Creates an event.

Prototype

```
EFI_STATUS
CreateEvent (
      IN UINT32                 Type,
      IN EFI_TPL                NotifyTpl,
      IN EFI_EVENT_NOTIFY       NotifyFunction,
      IN VOID                   *NotifyContext,
      OUT EFI_EVENT             *Event
);
```

Parameters

Type	The type of event to create and its mode and attributes. The **#define** statements in "Related Definitions" can be used to specify an event's mode and attributes.
NotifyTpl	The task priority level of event notifications. See **RaiseTPL()**.
NotifyFunction	Pointer to the event's notification function. See "Related Definitions."
NotifyContext	Pointer to the notification function's context; corresponds to parameter *Context* in the notification function.
Event	Pointer to the newly created event if the call succeeds; undefined otherwise.

Related Definitions

```
//*******************************************************
// EFI_EVENT
//*******************************************************
typedef VOID    *EFI_EVENT

//*******************************************************
// Event Types
//*******************************************************
// These types can be "ORed" together as needed - for example,
// EVT_TIMER might be "Ored" with EVT_NOTIFY_WAIT or
// EVT_NOTIFY_SIGNAL.
#define EVT_TIMER                         0x80000000
#define EVT_RUNTIME                       0x40000000
#define EVT_RUNTIME_CONTEXT               0x20000000
```

```
#define EVT_NOTIFY_WAIT                      0x00000100
#define EVT_NOTIFY_SIGNAL                    0x00000200

#define EVT_SIGNAL_EXIT_BOOT_SERVICES        0x00000201
#define EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE    0x60000202
```

EVT_TIMER	The event is a timer event and may be passed to **SetTimer()**. Note that timers only function during boot services time.
EVT_RUNTIME	The event is allocated from runtime memory. If an event is to be signaled after the call to **ExitBootServices()**, the event's data structure and notification function need to be allocated from runtime memory. For more information, see **SetVirtualAddressMap()** in Chapter 6.
EVT_RUNTIME_CONTEXT	The event's *NotifyContext* pointer points to a runtime memory address. See the discussion of **EVT_RUNTIME** above.
EVT_NOTIFY_WAIT	The event's *NotifyFunction* is to be invoked whenever the event is being waited on via **WaitForEvent()** or **CheckEvent()**.
EVT_NOTIFY_SIGNAL	The event's *NotifyFunction* is to be invoked whenever the event is signaled via **SignalEvent()**.
EVT_SIGNAL_EXIT_BOOT_SERVICES	This event is to be notified by the system when **ExitBootServices()** is invoked. This type cannot be used with any other EVT bit type. The notification function for this event is not allowed to use the Memory Allocation Services, or call any functions that use the Memory Allocation Services, because these services modify the current memory map.
EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE	The event is to be notified by the system when **SetVirtualAddressMap()** is performed. This type cannot be used with any other EVT bit type. See the discussion of **EVT_RUNTIME**.

```
//*******************************************************
// EFI_EVENT_NOTIFY
//*******************************************************
typedef
VOID
(EFIAPI *EFI_EVENT_NOTIFY) (
     IN EFI_EVENT               Event,
     IN VOID                    *Context
     );
```

Event	Event whose notification function is being invoked.
Context	Pointer to the notification function's context, which is implementation-dependent. *Context* corresponds to *NotifyContext* in **CreateEvent()**.

Description

The **CreateEvent()** function creates a new event of type *Type* and returns it in the location referenced by *Event*. The event's notification function, context, and task priority level are specified by *NotifyFunction*, *NotifyContext*, and *NotifyTpl*, respectively.

Events exist in one of two states, "waiting" or "signaled." When an event is created, firmware puts it in the "waiting" state. When the event is signaled, firmware changes its state to "signaled" and, if **EVT_NOTIFY_SIGNAL** is specified, places a call to its notification function in a FIFO queue. There is a queue for each of the "basic" task priority levels defined in Section 5.1 (**TPL_APPLICATION**, **TPL_CALLBACK**, and **TPL_NOTIFY**). The functions in these queues are invoked in FIFO order, starting with the highest priority level queue and proceeding to the lowest priority queue that is unmasked by the current TPL. If the current TPL is equal to or greater than the queued notification, it will wait until the TPL is lowered via **RestoreTPL()**.

In a general sense, there are two "types" of events, synchronous and asynchronous. Asynchronous events are closely related to timers and are used to support periodic or timed interruption of program execution. This capability is typically used with device drivers. For example, a network device driver that needs to poll for the presence of new packets could create an event whose type includes **EVT_TIMER** and then call the **SetTimer()** function. When the timer expires, the firmware signals the event.

Synchronous events have no particular relationship to timers. Instead, they are used to ensure that certain activities occur following a call to a specific interface function. One example of this is the cleanup that needs to be performed in response to a call to the **ExitBootServices()** function. **ExitBootServices()** can clean up the firmware since it understands firmware internals, but it cannot clean up on behalf of drivers that have been loaded into the system. The drivers have to do that themselves by creating an event whose type is **EVT_SIGNAL_EXIT_BOOT_SERVICES** and whose notification function is a function within the driver itself. Then, when **ExitBootServices()** has finished its cleanup, it signals each event of type **EVT_SIGNAL_EXIT_BOOT_SERVICES**.

Another example of the use of synchronous events occurs when an event of type **EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE** is used in conjunction with the **SetVirtualAddressMap()** function in Chapter 6.

The **EVT_NOTIFY_WAIT** and **EVT_NOTIFY_SIGNAL** flags are exclusive. If neither flag is specified, the caller does not require any notification concerning the event and the *NotifyTpl*, *NotifyFunction*, and *NotifyContext* parameters are ignored. If **EVT_NOTIFY_WAIT** is specified, then the event is signaled and its notify function is queued whenever a consumer of the event is waiting for it (via **WaitForEvent()** or **CheckEvent()**). If the **EVT_NOTIFY_SIGNAL** flag is specified then the event's notify function is queued whenever the event is signaled.

NOTE

Because its internal structure is unknown to the caller, Event cannot be modified by the caller. The only way to manipulate it is to use the published event interfaces.

Status Codes Returned

EFI_SUCCESS	The event structure was created.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_OUT_OF_RESOURCES	The event could not be allocated.

CloseEvent()

Summary

Closes an event.

Prototype

```
EFI_STATUS
CloseEvent (
      IN EFI_EVENT     Event
      );
```

Parameters

Event	The event to close. Type **EFI_EVENT** is defined in the **CreateEvent()** function description.

Description

The **CloseEvent()** function removes the caller's reference to the event and closes it. Once the event is closed, the event is no longer valid and may not be used on any subsequent function calls.

Status Codes Returned

EFI_SUCCESS	The event has been closed.

SignalEvent()

Summary

Signals an event.

Prototype

```
EFI_STATUS
SignalEvent (
    IN EFI_EVENT      Event
    );
```

Parameters

Event	The event to signal. Type **EFI_EVENT** is defined in the **CreateEvent()** function description.

Description

The supplied *Event* is signaled and, if the event has a signal notification function, it is scheduled to be invoked at the event's notification task priority level. **SignalEvent()** may be invoked from any task priority level.

Status Codes Returned

EFI_SUCCESS	The event was signaled.

WaitForEvent()

Summary

Stops execution until an event is signaled.

Prototype

```
EFI_STATUS
WaitForEvent (
     IN UINTN                      NumberOfEvents,
     IN EFI_EVENT                  *Event,
     OUT UINTN                     *Index
     );
```

Parameters

NumberOfEvents	The number of events in the *Event* array.
Event	An array of **EFI_EVENT**. Type **EFI_EVENT** is defined in the **CreateEvent()** function description.
Index	Pointer to the index of the event which satisfied the wait condition.

Description

The **WaitForEvent()** function waits for any event in the *Event* array to be signaled. On success, the signaled state of the event is cleared and execution is returned with *Index* indicating which event caused the return. It is possible for an event to be signaled before being waited on. In this case, the next wait operation for that event would immediately return with the signaled event.

Waiting on an event of type **EVT_NOTIFY_SIGNAL** is not permitted. If any event in *Event* is of type **EVT_NOTIFY_SIGNAL**, **WaitForEvent()** returns **EFI_INVALID_PARAMETER** and sets *Index* to indicate which event caused the failure. This function must be called at priority level **TPL_APPLICATION**. If an attempt is made to call it at any other priority level, **EFI_UNSUPPORTED** is returned.

To wait for a specified time, a timer event must be included in the *Event* array.

WaitForEvent() will always check for signaled events in order, with the first event in the array being checked first. To check if an event is signaled without waiting, an already signaled event can be used as the last event in the list being checked, or the **CheckEvent()** interface may be used.

Status Codes Returned

EFI_SUCCESS	The event indicated by *Index* was signaled.
EFI_INVALID_PARAMETER	The event indicated by *Index* has a notification function or *Event* was not a valid type.
EFI_UNSUPPORTED	The current TPL is not **TPL_APPLICATION**.

CheckEvent()

Summary

Checks whether an event is in the signaled state.

Prototype

```
EFI_STATUS
CheckEvent (
      IN EFI_EVENT     Event
      );
```

Parameters

Event	The event to check. Type **EFI_EVENT** is defined in the **CreateEvent()** function description.

Description

The **CheckEvent()** function checks to see whether *Event* is in the signaled state. If *Event* is of type **EVT_NOTIFY_SIGNAL**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, there are three possibilities:

- If *Event* is in the signaled state, it is cleared and **EFI_SUCCESS** is returned.
- If *Event* is not in the signaled state and has no notification function, **EFI_NOT_READY** is returned.
- If *Event* is not in the signaled state but does have a notification function, the function is executed. If that causes *Event* to be signaled, it is cleared and **EFI_SUCCESS** is returned; if it does not cause *Event* to be signaled, **EFI_NOT_READY** is returned.

Status Codes Returned

EFI_SUCCESS	The event is in the signaled state.
EFI_NOT_READY	The event is not in the signaled state.
EFI_INVALID_PARAMETER	*Event* is of type **EVT_NOTIFY_SIGNAL**.

SetTimer()

Summary

Sets the type of timer and the trigger time for a timer event.

Prototype

```
EFI_STATUS
SetTimer (
      IN EFI_EVENT           Event,
      IN EFI_TIMER_DELAY     Type,
      IN UINT64              TriggerTime
      );
```

Parameters

Event	The timer event that is to be signaled at the specified time. Type **EFI_EVENT** is defined in the **CreateEvent()** function description.
Type	The type of time that is specified in *TriggerTime*. See the timer delay types in “Related Definitions.”
TriggerTime	The number of 100ns units until the timer expires.

Related Definitions

```
//*******************************************************
//EFI_TIMER_DELAY
//*******************************************************
typedef enum {
      TimerCancel,
      TimerPeriodic,
      TimerRelative
} EFI_TIMER_DELAY;
```

TimerCancel	The event’s timer setting is to be cancelled and no timer trigger is to be set. *TriggerTime* is ignored when canceling a timer.
TimerPeriodic	The event is to be signaled periodically at *TriggerTime* intervals from the current time. This is the only timer trigger *Type* for which the event timer does not need to be reset for each notification. All other timer trigger types are “one shot.”
TimerRelative	The event is to be signaled in *TriggerTime* 100ns units.

Description

The **SetTimer()** function cancels any previous time trigger setting for the event, and sets the new trigger time for the event. This function can only be used on events of type **EVT_TIMER**.

Status Codes Returned

EFI_SUCCESS	The event has been set to be signaled at the requested time.
EFI_INVALID_PARAMETER	*Event* or *Type* is not valid.

RaiseTPL()

Summary

Raises a task's priority level and returns its previous level.

Prototype

```
EFI_TPL
RaiseTPL (
      IN EFI_TPL NewTpl
      );
```

Parameters

NewTpl	The new task priority level. It must be greater than or equal to the current task priority level. See "Related Definitions."

Related Definitions

```
//*******************************************************
// EFI_TPL
//*******************************************************
typedef UINTN    EFI_TPL

//*******************************************************
// Task Priority Levels
//*******************************************************
#define TPL_APPLICATION     4
#define TPL_CALLBACK        8
#define TPL_NOTIFY          16
#define TPL_HIGH_LEVEL      31
```

Description

The **RaiseTPL()** function raises the priority of the currently executing task and returns its previous priority level.

Only three task priority levels are exposed outside of the firmware during EFI boot services execution. The first is **TPL_APPLICATION** where all normal execution occurs. That level may be interrupted to perform various asynchronous interrupt style notifications, which occur at the **TPL_CALLBACK** or **TPL_NOTIFY** level. By raising the task priority level to **TPL_NOTIFY** such notifications are masked until the task priority level is restored, thereby synchronizing execution with such notifications. Synchronous blocking I/O functions execute at **TPL_NOTIFY**. **TPL_CALLBACK** is the typically used for application level notification functions. Device drivers will typically use **TPL_CALLBACK** or **TPL_NOTIFY** for their notification functions. Applications and drivers may also use **TPL_NOTIFY** to protect data structures in critical sections of code.

The caller must restore the task priority level with **RestoreTPL()** to the previous level before returning.

NOTE

If NewTpl is below the current TPL level, then the system behavior is indeterminate. Additionally, only **TPL_APPLICATION**, **TPL_CALLBACK**, **TPL_NOTIFY**, *and* **TPL_HIGH_LEVEL** *may be used. All other values are reserved for use by the firmware; using them will result in unpredictable behavior. Good coding practice dictates that all code should execute at its lowest possible TPL level, and the use of TPL levels above* **TPL_APPLICATION** *must be minimized. Executing at TPL levels above* **TPL_APPLICATION** *for extended periods of time may also result in unpredictable behavior.*

Status Codes Returned

Unlike other EFI interface functions, **RaiseTPL()** does not return a status code. Instead, it returns the previous task priority level, which is to be restored later with a matching call to **RestoreTPL()**.

RestoreTPL()

Summary

Restores a task's priority level to its previous value.

Prototype

```
VOID
RestoreTPL (
      IN EFI_TPL OldTpl
      )
```

Parameters

OldTpl	The previous task priority level to restore (the value from a previous, matching call to **RaiseTPL()**). Type **EFI_TPL** is defined in the **RaiseTPL()** function description.

Description

The **RestoreTPL()** function restores a task's priority level to its previous value. Calls to **RestoreTPL()** are matched with calls to **RaiseTPL()**.

NOTE

If OldTpl is above the current TPL level, then the system behavior is indeterminate. Additionally, only ***TPL_APPLICATION***, ***TPL_CALLBACK***, ***TPL_NOTIFY***, *and* **TPL_HIGH_LEVEL** *may be used. All other values are reserved for use by the firmware; using them will result in unpredictable behavior. Good coding practice dictates that all code should execute at its lowest possible TPL level, and the use of TPL levels above* **TPL_APPLICATION** *must be minimized. Executing at TPL levels above* **TPL_APPLICATION** *for extended periods of time may also result in unpredictable behavior.*

Status Codes Returned

None.

5.2 Memory Allocation Services

The functions that make up Memory Allocation Services are used during preboot to allocate and free memory, and to obtain the system's memory map. See Table 5-4.

Table 5-4. Memory Allocation Functions

Name	Type	Description
AllocatePages	Boot	Allocates pages of a particular type.
FreePages	Boot	Frees allocated pages.
GetMemoryMap	Boot	Returns the current boot services memory map and memory map key.
AllocatePool	Boot	Allocates a pool of a particular type.
FreePool	Boot	Frees allocated pool.

The way in which these functions are used is directly related to an important feature of EFI memory design. This feature, which stipulates that EFI firmware owns the system's memory map during preboot, has three major consequences:

1. During preboot, all components (including executing EFI images) must cooperate with the firmware by allocating and freeing memory from the system with the functions **AllocatePages()**, **AllocatePool()**, **FreePages()**, and **FreePool()**. The firmware dynamically maintains the memory map as these functions are called.
2. During preboot, an executing EFI Image must only use the memory it has allocated.
3. Before an executing EFI image exits and returns control to the firmware, it must free all resources it has explicitly allocated. This includes all memory pages, pool allocations, open file handles, etc. Memory allocated by the firmware to load an image is freed by the firmware when the image is unloaded.

When EFI memory is allocated, it is "typed" according to the values in **EFI_MEMORY_TYPE** (see the description for **AllocatePages()**). Some of the types have a different usage *before* **ExitBootServices()** is called than they do *afterwards*. Table 5-5 lists each type and its usage before the call; Table 5-6 lists each type and its usage after the call. The system firmware must follow the processor-specific rules outlined in sections 2.3.2 and 2.3.3 in the layout of the EFI memory map to enable the OS to make the required virtual mappings.

Table 5-5. Memory Type Usage before `ExitBootServices()`

Mnemonic	Description
EfiReservedMemoryType	Not used.
EfiLoaderCode	The code portions of a loaded EFI application. (Note that EFI OS loaders are EFI applications.)
EfiLoaderData	The data portions of a loaded EFI application and the default data allocation type used by an EFI application to allocate pool memory.
EfiBootServicesCode	The code portions of a loaded Boot Services Driver.
EfiBootServicesData	The data portions of a loaded Boot Serves Driver, and the default data allocation type used by a Boot Services Driver to allocate pool memory.
EfiRuntimeServicesCode	The code portions of a loaded Runtime Services Driver.
EfiRuntimeServicesData	The data portions of a loaded Runtime Services Driver and the default data allocation type used by a Runtime Services Driver to allocate pool memory.
EfiConventionalMemory	Free (unallocated) memory.
EfiUnusableMemory	Memory in which errors have been detected.
EfiACPIReclaimMemory	Memory that holds the ACPI tables.
EfiACPIMemoryNVS	Address space reserved for use by the firmware.
EfiMemoryMappedIO	Used by system firmware to request that a memory-mapped IO region be mapped by the OS to a virtual address so it can be accessed by EFI runtime services.
EfiMemoryMappedIOPortSpace	System memory-mapped IO region that is used to translate memory cycles to IO cycles by the processor.
EfiPalCode	Address space reserved by the firmware for code that is part of the processor.

NOTE

There is only one region of type **`EfiMemoryMappedIoPortSpace`** *defined in the architecture for Itanium-based platforms. As a result, there should be one and only one region of type* **`EfiMemoryMappedIoPortSpace`** *in the EFI memory map of an Itanium-based platform.*

Table 5-6. Memory Type Usage after ExitBootServices()

Mnemonic	Description
EfiReservedMemoryType	Not used.
EfiLoaderCode	The Loader and/or OS may use this memory as they see fit. Note: the OS loader that called **ExitBootServices()** is utilizing one or more **EfiLoaderCode** ranges.
EfiLoaderData	The Loader and/or OS may use this memory as they see fit. Note: the OS loader that called **ExitBootServices()** is utilizing one or more **EfiLoaderData** ranges.
EfiBootServicesCode	Memory available for general use.
EfiBootServicesData	Memory available for general use.
EfiRuntimeServicesCode	The memory in this range is to be preserved by the loader and OS in the working and ACPI S1–S3 states.
EfiRuntimeServicesData	The memory in this range is to be preserved by the loader and OS in the working and ACPI S1–S3 states.
EfiConventionalMemory	Memory available for general use.
EfiUnusableMemory	Memory that contains errors and is not to be used.
EfiACPIReclaimMemory	This memory is to be preserved by the loader and OS until ACPI is enabled. Once ACPI is enabled, the memory in this range is available for general use.
EfiACPIMemoryNVS	This memory is to be preserved by the loader and OS in the working and ACPI S1–S3 states.
EfiMemoryMappedIO	This memory is not used by the OS. All system memory-mapped IO information should come from ACPI tables.
EfiMemoryMappedIOPortSpace	This memory is not used by the OS. All system memory-mapped IO port space information should come from ACPI tables.
EfiPalCode	This memory is to be preserved by the loader and OS in the working and ACPI S1–S3 states. This memory may also have other attributes that are defined by the processor implementation.

NOTE

An image that calls **ExitBootServices**() *first calls* **GetMemoryMap()** *to obtain the current memory map. Following the* **ExitBootServices**() *call, the image implicitly owns all* unused *memory in the map. This includes memory types EfiLoaderCode, EfiLoaderData, EfiBootServicesCode, EfiBootServicesData, and EfiConventionalMemory. An EFI-compatible loader and operating system must preserve the memory marked as EfiRuntimeServicesCode and EfiRuntimeServicesData.*

AllocatePages()

Summary

Allocates memory pages from the system.

Prototype

```
EFI_STATUS
AllocatePages(
      IN EFI_ALLOCATE_TYPE             Type,
      IN EFI_MEMORY_TYPE               MemoryType,
      IN UINTN                         Pages,
      IN OUT EFI_PHYSICAL_ADDRESS *Memory
      );
```

Parameters

Type	The type of allocation to perform. See "Related Definitions."
MemoryType	The type of memory to allocate. The type **EFI_MEMORY_TYPE** is defined in "Related Definitions" below. These memory types are also described in more detail in Table 5-5 and Table 5-6. Normal allocations (that is, allocations by any EFI application) are of type **EfiLoaderData**. *MemoryType* values in the range 0x80000000..0xFFFFFFFF are reserved for use by EFI OS loaders that are provided by operating system vendors. The only illegal memory type values are those in the range **EfiMaxMemoryType**..0x7FFFFFFF.
Pages	The number of contiguous 4 KB pages to allocate.
Memory	Pointer to a physical address. On input, the way in which the address is used depends on the value of *Type*. See "Description" for more information. On output the address is set to the base of the page range that was allocated. See "Related Definitions."

Related Definitions

```
//*******************************************************
//EFI_ALLOCATE_TYPE
//*******************************************************
// These types are discussed in the “Description” section below.
 typedef enum {
     AllocateAnyPages,
     AllocateMaxAddress,
     AllocateAddress,
     MaxAllocateType
 } EFI_ALLOCATE_TYPE;

//*******************************************************
//EFI_MEMORY_TYPE
//*******************************************************
// These type values are discussed in Table 5-5 and Table 5-6.
 typedef enum {
     EfiReservedMemoryType,
     EfiLoaderCode,
     EfiLoaderData,
     EfiBootServicesCode,
     EfiBootServicesData,
     EfiRuntimeServicesCode,
     EfiRuntimeServicesData,
     EfiConventionalMemory,
     EfiUnusableMemory,
     EfiACPIReclaimMemory,
     EfiACPIMemoryNVS,
     EfiMemoryMappedIO,
     EfiMemoryMappedIOPortSpace,
     EfiPalCode,
     EfiMaxMemoryType
 } EFI_MEMORY_TYPE;

//*******************************************************
//EFI_PHYSICAL_ADDRESS
//*******************************************************
typedef UINT64        EFI_PHYSICAL_ADDRESS;
```

Description

The **AllocatePages()** function allocates the requested number of pages and returns a pointer to the base address of the page range in the location referenced by *Memory*. The function scans the memory map to locate free pages. When it finds a physically contiguous block of pages that is large enough and also satisfies the value of *Type*, it changes the memory map to indicate that the pages are now of type *MemoryType*.

In general, EFI OS loaders and EFI applications should allocate memory (and pool) of type **EfiLoaderData**. Boot service drivers must allocate memory (and pool) of type **EfiBootServicesData**. Runtime drivers should allocate memory (and pool) of type **EfiRuntimeServicesData** (although such allocation can only be made during boot services time).

Allocation requests of *Type* **AllocateAnyPages** allocate any available range of pages that satisfies the request. On input, the address pointed to by *Memory* is ignored.

Allocation requests of *Type* **AllocateMaxAddress** allocate any available range of pages whose uppermost address is less than or equal to the address pointed to by *Memory* on input.

Allocation requests of *Type* **AllocateAddress** allocate pages at the address pointed to by *Memory* on input.

Status Codes Returned

EFI_SUCCESS	The requested pages were allocated.
EFI_OUT_OF_RESOURCES	The pages could not be allocated.
EFI_INVALID_PARAMETER	*Type* is not **AllocateAnyPages** or **AllocateMaxAddress** or **AllocateAddress**.
EFI_INVALID_PARAMETER	*MemoryType* is in the range **EfiMaxMemoryType**..0x7FFFFFFF.
EFI_NOT_FOUND	The requested pages could not be found.

FreePages()

Summary

Frees memory pages.

Prototype

```
EFI_STATUS
FreePages (
       IN EFI_PHYSICAL_ADDRESS    Memory,
       IN UINTN                   Pages
       );
```

Parameters

Memory	The base physical address of the pages to be freed. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description.
Pages	The number of contiguous 4 KB pages to free.

Description

The **FreePages()** function returns memory allocated by **AllocatePages()** to the firmware.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were freed.
EFI_NOT_FOUND	The requested memory pages were not allocated with **AllocatePages()**.
EFI_INVALID_PARAMETER	*Memory* is not a page-aligned address or *Pages* is invalid.

GetMemoryMap()

Summary

Returns the current memory map.

Prototype

```
EFI_STATUS
GetMemoryMap (
     IN OUT UINTN                    *MemoryMapSize,
     IN OUT EFI_MEMORY_DESCRIPTOR    *MemoryMap,
     OUT UINTN                       *MapKey,
     OUT UINTN                       *DescriptorSize,
     OUT UINT32                      *DescriptorVersion
     );
```

Parameters

MemoryMapSize	A pointer to the size, in bytes, of the *MemoryMap* buffer. On input, this is the size of the buffer allocated by the caller. On output, it is the size of the buffer returned by the firmware if the buffer was large enough, or the size of the buffer needed to contain the map if the buffer was too small.
MemoryMap	A pointer to the buffer in which firmware places the current memory map. The map is an array of **EFI_MEMORY_DESCRIPTOR**s. See “Related Definitions.”
MapKey	A pointer to the location in which firmware returns the key for the current memory map.
DescriptorSize	A pointer to the location in which firmware returns the size, in bytes, of an individual **EFI_MEMORY_DESCRIPTOR**.
DescriptorVersion	A pointer to the location in which firmware returns the version number associated with the **EFI_MEMORY_DESCRIPTOR**. See “Related Definitions.”

Related Definitions

```
//*******************************************************
//EFI_MEMORY_DESCRIPTOR
//*******************************************************
typedef struct {
UINT32                  Type;
EFI_PHYSICAL_ADDRESS    PhysicalStart;
EFI_VIRTUAL_ADDRESS     VirtualStart;
UINT64                  NumberOfPages;
UINT64                  Attribute;
} EFI_MEMORY_DESCRIPTOR;
```

Type	Type of the memory region. Type **EFI_MEMORY_TYPE** is defined in the **AllocatePages()** function description.
PhysicalStart	Physical address of the first byte in the memory region. Physical start must be aligned on a 4 KB boundary. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description.
VirtualStart	Virtual address of the first byte in the memory region. Virtual start must be aligned on a 4 KB boundary. Type **EFI_VIRTUAL_ADDRESS** is defined in "Related Definitions."
NumberOfPages	Number of 4 KB pages in the memory region.
Attribute	Attributes of the memory region that describe the bit mask of capabilities for that memory region, and not necessarily the current settings for that memory region. See the following "Memory Attribute Definitions."

```
//*******************************************************
// Memory Attribute Definitions
//*******************************************************
// These types can be "ORed" together as needed.
#define EFI_MEMORY_UC           0x0000000000000001
#define EFI_MEMORY_WC           0x0000000000000002
#define EFI_MEMORY_WT           0x0000000000000004
#define EFI_MEMORY_WB           0x0000000000000008
#define EFI_MEMORY_UCE          0x0000000000000010
#define EFI_MEMORY_WP           0x0000000000001000
#define EFI_MEMORY_RP           0x0000000000002000
#define EFI_MEMORY_XP           0x0000000000004000
#define EFI_MEMORY_RUNTIME      0x8000000000000000
```

EFI_MEMORY_UC	Memory cacheability attribute: The memory region supports being configured as not cacheable.
EFI_MEMORY_WC	Memory cacheability attribute: The memory region supports being configured as write combining.
EFI_MEMORY_WT	Memory cacheability attribute: The memory region supports being configured as cacheable with a "write through" policy. Writes that hit in the cache will also be written to main memory.
EFI_MEMORY_WB	Memory cacheability attribute: The memory region supports being configured as cacheable with a "write back" policy. Reads and writes that hit in the cache do not propagate to main memory. Dirty data is written back to main memory when a new cache line is allocated.
EFI_MEMORY_UCE	Memory cacheability attribute: The memory region supports being configured as not cacheable, exported, and supports the "fetch and add" semaphore mechanism.
EFI_MEMORY_WP	Physical memory protection attribute: The memory region supports being configured as write-protected by system hardware.
EFI_MEMORY_RP	Physical memory protection attribute: The memory region supports being configured as read-protected by system hardware.
EFI_MEMORY_XP	Physical memory protection attribute: The memory region supports being configured so it is protected by system hardware from executing code.
EFI_MEMORY_RUNTIME	Runtime memory attribute: The memory region needs to be given a virtual mapping by the operating system when **SetVirtualAddressMap()** is called (described in Chapter 6).

```
//*******************************************************
//EFI_VIRTUAL_ADDRESS
//*******************************************************
typedef UINT64        EFI_VIRTUAL_ADDRESS;

//*******************************************************
// Memory Descriptor Version Number
//*******************************************************
#define EFI_MEMORY_DESCRIPTOR_VERSION  1
```

Description

The **GetMemoryMap()** function returns a copy of the current memory map. The map is an array of memory descriptors, each of which describes a contiguous block of memory. The map describes all of memory, no matter how it is being used. That is, it includes blocks allocated by **AllocatePages()** and **AllocatePool()**, as well as blocks that the firmware is using for its own purposes. The memory map is only used to describe memory that is present in the system. Memory descriptors are never used to describe holes in the system memory map.

Until **ExitBootServices()** is called, the memory map is owned by the firmware and the currently executing EFI Image should only use memory pages it has explicitly allocated.

If the *MemoryMap* buffer is too small, the **EFI_BUFFER_TOO_SMALL** error code is returned and the *MemoryMapSize* value contains the size of the buffer needed to contain the current memory map.

On success a *MapKey* is returned that identifies the current memory map. The firmware's key is changed every time something in the memory map changes. In order to successfully invoke **ExitBootServices()** the caller must provide the current memory map key.

The **GetMemoryMap()** function also returns the size and revision number of the **EFI_MEMORY_DESCRIPTOR**. The *DescriptorSize* represents the size in bytes of an **EFI_MEMORY_DESCRIPTOR** array element returned in *MemoryMap*. The size is returned to allow for future expansion of the **EFI_MEMORY_DESCRIPTOR** in response to hardware innovation. The structure of the **EFI_MEMORY_DESCRIPTOR** may be extended in the future but it will remain backwards compatible with the current definition. Thus OS software must use the *DescriptorSize* to find the start of each **EFI_MEMORY_DESCRIPTOR** in the *MemoryMap* array.

Status Codes Returned

EFI_SUCCESS	The memory map was returned in the *MemoryMap* buffer.
EFI_BUFFER_TOO_SMALL	The *MemoryMap* buffer was too small. The current buffer size needed to hold the memory map is returned in *MemoryMapSize*.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

AllocatePool()

Summary

Allocates pool memory.

Prototype

```
EFI_STATUS
AllocatePool (
     IN EFI_MEMORY_TYPE          PoolType,
     IN UINTN                    Size,
     OUT VOID                    **Buffer
     );
```

Parameters

PoolType	The type of pool to allocate. Type **EFI_MEMORY_TYPE** is defined in the **AllocatePages()** function description. *PoolType* values in the range 0x80000000..0xFFFFFFFF are reserved for use by EFI OS loaders that are provided by operating system vendors. The only illegal memory type values are those in the range **EfiMaxMemoryType**..0x7FFFFFFF.
Size	The number of bytes to allocate from the pool.
Buffer	A pointer to a pointer to the allocated buffer if the call succeeds; undefined otherwise.

Description

The **AllocatePool()** function allocates a memory region of *Size* bytes from memory of type *PoolType* and returns the address of the allocated memory in the location referenced by *Buffer*. This function allocates pages from **EfiConventionalMemory** as needed to grow the requested pool type. All allocations are eight-byte aligned.

The allocated pool memory is returned to the available pool with the **FreePool()** function.

Status Codes Returned

EFI_SUCCESS	The requested number of bytes was allocated.
EFI_OUT_OF_RESOURCES	The pool requested could not be allocated.
EFI_INVALID_PARAMETER	*PoolType* was invalid.

FreePool()

Summary

Returns pool memory to the system.

Prototype

```
EFI_STATUS
FreePool (
          IN VOID   *Buffer
          );
```

Parameters

Buffer	Pointer to the buffer to free.

Description

The **FreePool()** function returns the memory specified by *Buffer* to the system. On return, the memory's type is **EfiConventionalMemory**. The *Buffer* that is freed must have been allocated by **AllocatePool()**.

Status Codes Returned

EFI_SUCCESS	The memory was returned to the system.
EFI_INVALID_PARAMETER	*Buffer* was invalid.

5.3 Protocol Handler Services

In the abstract, a protocol consists of a 128-bit globally unique identifier (GUID) and a Protocol Interface structure. The structure contains the functions and instance data that are used to access a device. The functions that make up Protocol Handler Services allow applications to install a protocol on a handle, identify the handles that support a given protocol, determine whether a handle supports a given protocol, and so forth. See Table 5-7.

Table 5-7. Protocol Interface Functions

Name	Type	Description
InstallProtocolInterface	Boot	Installs a protocol interface on a device handle.
UninstallProtocolInterface	Boot	Removes a protocol interface from a device handle.
ReinstallProtocolInterface	Boot	Reinstalls a protocol interface on a device handle.
RegisterProtocolNotify	Boot	Registers an event that is to be signaled whenever an interface is installed for a specified protocol.
LocateHandle	Boot	Returns an array of handles that support a specified protocol.
HandleProtocol	Boot	Queries a handle to determine if it supports a specified protocol.
LocateDevicePath	Boot	Locates all devices on a device path that support a specified protocol and returns the handle to the device that is closest to the path.
OpenProtocol	Boot	Adds elements to the list of agents consuming a protocol interface.
CloseProtocol	Boot	Removes elements from the list of agents consuming a protocol interface.
OpenProtocolInformation	Boot	Retrieve the list of agents that are currently consuming a protocol interface.
ConnectController	Boot	Uses a set of precedence rules to find the best set of drivers to manage a controller.
DisconnectController	Boot	Informs a set of drivers to stop managing a controller.
ProtocolsPerHandle	Boot	Retrieves the list of protocols installed on a handle. The return buffer is automatically allocated.
LocateHandleBuffer	Boot	Retrieves the list of handles from the handle database that meet the search criteria. The return buffer is automatically allocated.
LocateProtocol	Boot	Finds the first handle in the handle database the supports the requested protocol.
InstallMultipleProtocolInterfaces	Boot	Installs one or more protocol interfaces onto a handle.
UninstallMultipleProtocolInterfaces	Boot	Uninstalls one or more protocol interfaces from a handle.

The Protocol Handler boot services have been modified to take advantage of the information that is now being tracked with the **OpenProtocol()** and **CloseProtocol()** boot services. Since the usage of protocol interfaces is being tracked with these new boot services, it is now possible to safely uninstall and reinstall protocol interfaces that are being consumed by EFI drivers.

As depicted in Figure 5-1, the firmware is responsible for maintaining a "data base" that shows which protocols are attached to each device handle. (The figure depicts the "data base" as a linked list, but the choice of data structure is implementation-dependent.) The "data base" is built dynamically by calling the **InstallProtocolInterface()** function. Protocols can only be installed by EFI drivers or the firmware itself. In the figure, a device handle (**EFI_HANDLE**) refers to a list of one or more registered protocol interfaces for that handle. The first handle in the system has four attached protocols, and the second handle has two attached protocols. Each attached protocol is represented as a GUID/Interface pointer pair. The GUID is the name of the protocol, and Interface points to a protocol instance. This data structure will typically contain a list of interface functions, and some amount of instance data.

Access to devices is initiated by calling the **HandleProtocol()** function, which determines whether a handle supports a given protocol. If it does, a pointer to the matching Protocol Interface structure is returned.

When a protocol is added to the system, it may either be added to an existing device handle or it may be added to create a new device handle. Figure 5-1 shows that protocol handlers are listed for each device handle and that each protocol handler is logically an EFI driver.

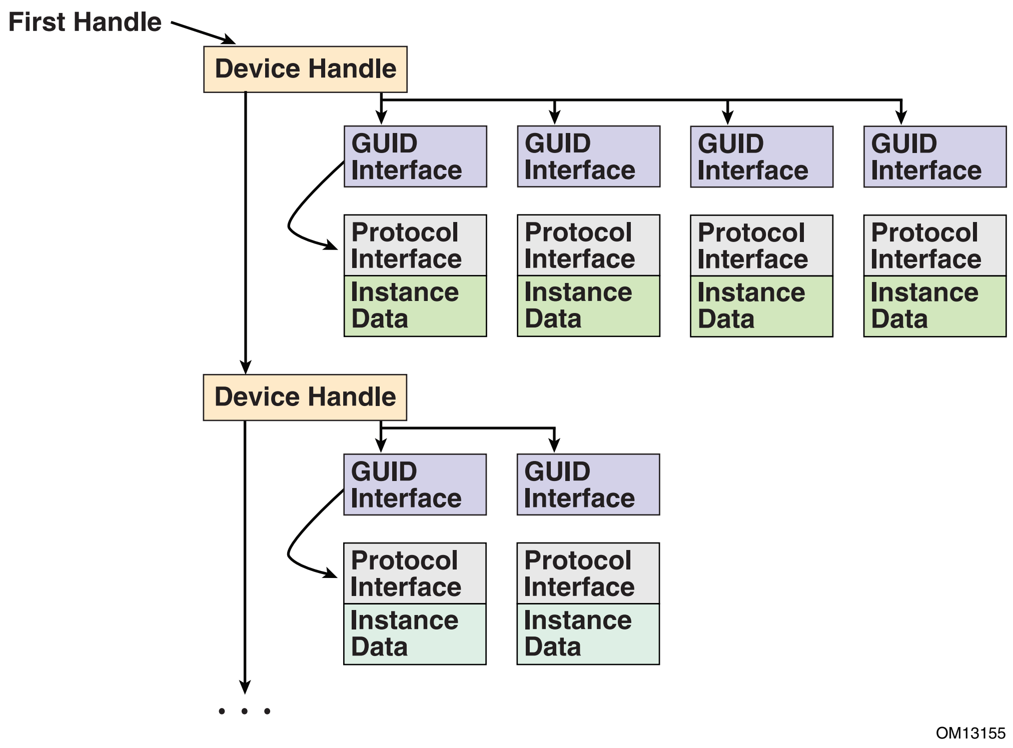


Figure 5-1. Device Handle to Protocol Handler Mapping

The ability to add new protocol interfaces as new handles or to layer them on existing interfaces provides great flexibility. Layering makes it possible to add a new protocol that builds on a device's basic protocols. An example of this might be to layer on a **`SIMPLE_TEXT_OUTPUT`** protocol support that would build on the handle's underlying **`SERIAL_IO`** protocol.

The ability to add new handles can be used to generate new devices as they are found, or even to generate abstract devices. An example of this might be to add a multiplexing device that replaces *`ConsoleOut`* with a virtual device that multiplexes the **`SIMPLE_TEXT_OUTPUT`** protocol onto multiple underlying device handles.

5.3.1 Driver Model Boot Services

This section provides a detailed description of the new EFI boot services that are required by the *EFI Driver Model*. These boot services are being added to reduce the size and complexity of the bus drivers and device drivers. This, in turn, will reduce the amount of ROM space required by drivers that are programmed into ROMs on adapters or into system FLASH, and reduce the development and testing time required by driver writers.

These new services fall into two categories. The first group is used to track the usage of protocol interfaces by different agents in the system. Protocol interfaces are stored in a handle database. The handle database consists of a list of handles, and on each handle there is a list of one or more protocol interfaces. The boot services **`InstallProtocolInterface()`**, **`UninstallProtocolInterface()`**, and **`ReinstallProtocolInterface()`** are used to add, remove, and replace protocol interfaces in the handle database. The boot service **`HandleProtocol()`** is used to look up a protocol interface in the handle database. However, agents that call **`HandleProtocol()`** are not tracked, so it is not safe to call **`UninstallProtocolInterface()`** or **`ReinstallProtocolInterface()`** because an agent may be using the protocol interface that is being removed or replaced.

The solution is to track the usage of protocol interfaces in the handle database itself. To accomplish this, each protocol interface includes a list of agents that are consuming the protocol interface. Figure 5-2 shows an example handle database with these new agent lists. An agent consists of an image handle, a controller handle, and some attributes. The image handle identifies the driver or application that is consuming the protocol interface. The controller handle identifies the controller that is consuming the protocol interface. Since a driver may manage more than one controller, the combination of a driver's image handle and a controller's controller handle uniquely identifies the agent that is consuming the protocol interface. The attributes show how the protocol interface is being used.

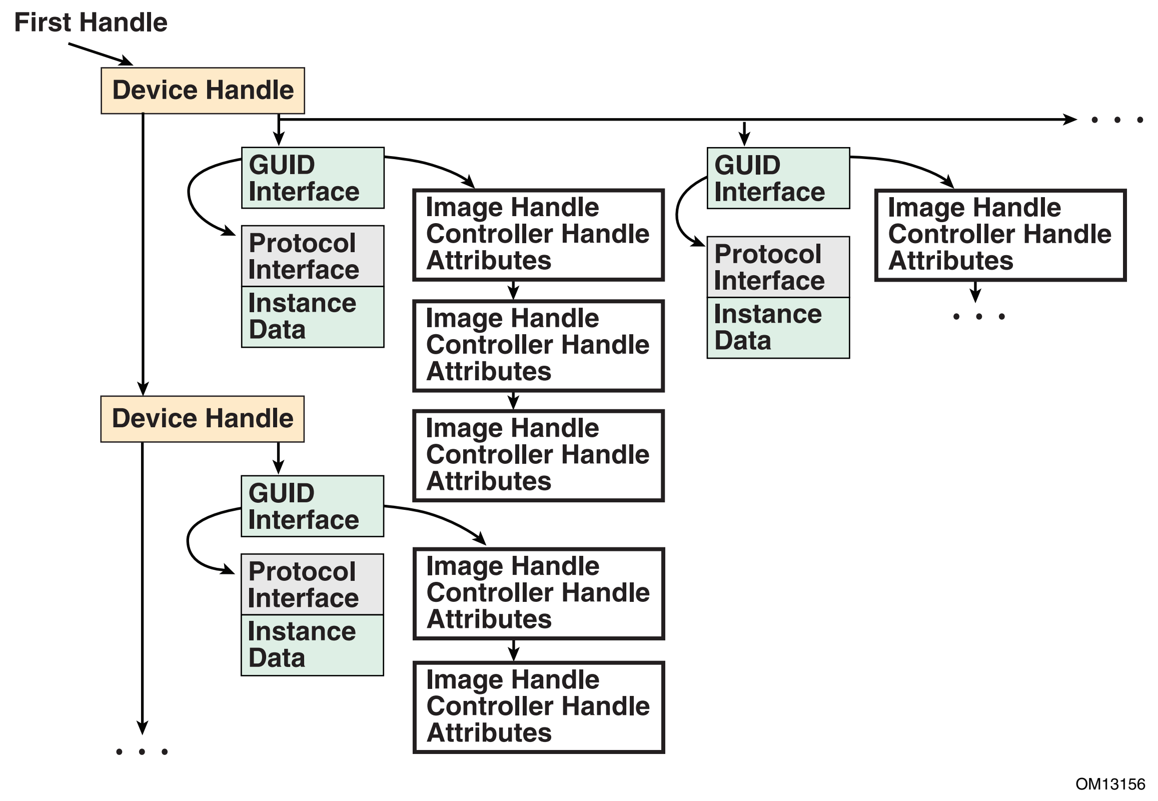

First Handle
Device Handle
GUID Interface
Protocol Interface
Instance Data
Image Handle Controller Handle Attributes
Image Handle Controller Handle Attributes
Image Handle Controller Handle Attributes
GUID Interface
Protocol Interface
Instance Data
Image Handle Controller Handle Attributes
Device Handle
GUID Interface
Protocol Interface
Instance Data
Image Handle Controller Handle Attributes
Image Handle Controller Handle Attributes
OM13156

Figure 5-2. Handle Database

In order to maintain these agent lists in the handle database, some new boot services are required. These are `OpenProtocol()`, `CloseProtocol()`, and `OpenProtocolInformation()`. `OpenProtocol()` adds elements to the list of agents consuming a protocol interface. `CloseProtocol()` removes elements from the list of agents consuming a protocol interface, and `OpenProtocolInformation()` retrieves the entire list of agents that are currently using a protocol interface.

The second group of boot services is used to deterministically connect and disconnect drivers to controllers. The boot services in this group are `ConnectController()` and `DisconnectController()`. These services take advantage of the new features of the handle database along with the new protocols described in this document to manage the drivers and controllers present in the system. `ConnectController()` uses a set of strict precedence rules to find the best set of drivers for a controller. This provides a deterministic matching of drivers to controllers with extensibility mechanisms for OEMs, IBVs, and IHVs. `DisconnectController()` allows drivers to be disconnected from controllers in a controlled manner, and by using the new features of the handle database it is possible to fail a disconnect request because a protocol interface cannot be released at the time of the disconnect request.

The third group of boot services is designed to help simplify the implementation of drivers, and produce drivers with smaller executable footprints. The `LocateHandleBuffer()` is a new version of `LocateHandle()` that allocates the required buffer for the caller. This eliminates two calls to `LocateHandle()` and a call to `AllocatePool()` from the caller's code. `LocateProtocol()` searches the handle database for the first protocol instance that matches the search criteria. The `InstallMultipleProtocolInterfaces()` and `UninstallMutipleProtocolInterfaces()` are very useful to driver writers. These boot services allow one or more protocol interfaces to be added or removed from a handle. In addition, `InstallMultipleProtocolInterfaces()` guarantees that a duplicate device path is never added to the handle database. This is very useful to bus drivers that can create one child handle at a time, because it guarantees that the bus driver will not inadvertently create two instances of the same child handle.

InstallProtocolInterface()

Summary

Installs a protocol interface on a device handle. If the handle does not exist, it is created and added to the list of handles in the system.

Prototype

```
EFI_STATUS
InstallProtocolInterface (
      IN OUT EFI_HANDLE                   *Handle,
      IN EFI_GUID                         *Protocol,
      IN EFI_INTERFACE_TYPE               InterfaceType,
      IN VOID                             *Interface
      );
```

Parameters

Handle	A pointer to the **EFI_HANDLE** on which the interface is to be installed. If **Handle* is **NULL** on input, a new handle is created and returned on output. If **Handle* is not **NULL** on input, the protocol is added to the handle, and the handle is returned unmodified. The type **EFI_HANDLE** is defined in "Related Definitions." If **Handle* is not a valid handle, then **EFI_INVALID_PARAMETER** is returned.
Protocol	The numeric ID of the protocol interface. The type **EFI_GUID** is defined in "Related Definitions." It is the callers responsibility to pass in a valid GUID. See "Wired For Management Baseline" for a description of valid GUID values.
InterfaceType	Indicates whether *Interface* is supplied in native form. This value indicates the original execution environment of the request. See "Related Definitions."
Interface	A pointer to the protocol interface. The *Interface* must adhere to the structure defined by *Protocol*. **NULL** can be used if a structure is not associated with *Protocol*.

Related Definitions

```
//********************************************************
//EFI_HANDLE
//********************************************************
typedef VOID          *EFI_HANDLE;

//********************************************************
//EFI_GUID
//********************************************************
typedef struct {
    UINT32  Data1;
    UINT16  Data2;
    UINT16  Data3;
    UINT8   Data4[8];
} EFI_GUID;

//********************************************************
//EFI_INTERFACE_TYPE
//********************************************************
typedef enum {
     EFI_NATIVE_INTERFACE
} EFI INTERFACE_TYPE;
```

Description

The **`InstallProtocolInterface()`** function installs a protocol interface (a GUID/Protocol Interface structure pair) on a device handle. The same GUID cannot be installed more than once onto the same handle. If the same GUID is installed more than once onto the same handle, then the results are not predictable.

Installing a protocol interface allows other components to locate the *`Handle`*, and the interfaces installed on it. A protocol interface is always installed at the head of the device handle's queue.

When a protocol interface is installed, the firmware calls all notification functions that have registered to wait for the installation of *`Protocol`*. For more information, see the **`RegisterProtocolNotify()`** function description.

Status Codes Returned

EFI_SUCCESS	The protocol interface was installed.
EFI_OUT_OF_RESOURCES	Space for a new handle could not be allocated.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

UninstallProtocolInterface()

Summary

Removes a protocol interface from a device handle.

Prototype

```
typedef
EFI_STATUS
UninstallProtocolInterface (
  IN EFI_HANDLE   Handle,
  IN EFI_GUID     *Protocol,
  IN VOID         *Interface
  );
```

Parameters

Handle	The handle on which the interface was installed. If *Handle* is not a valid handle, then **EFI_INVALID_PARAMETER** is returned. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description.
Protocol	The numeric ID of the interface. It is the caller's responsibility to pass in a valid GUID. See "Wired For Management Baseline" for a description of valid GUID values. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.
Interface	A pointer to the interface. **NULL** can be used if a structure is not associated with *Protocol*.

Description

The **UninstallProtocolInterface()** function removes a protocol interface from the handle on which it was previously installed. The *Protocol* and *Interface* values define the protocol interface to remove from the handle.

The caller is responsible for ensuring that there are no references to a protocol interface that has been removed. In some cases, outstanding reference information is not available in the protocol, so the protocol, once added, cannot be removed. Examples include Console I/O, Block I/O, Disk I/O, and (in general) handles to device protocols.

If the last protocol interface is removed from a handle, the handle is freed and is no longer valid.

EFI 1.10 Extension

The extension to this service directly addresses the limitations described in the section above. There may be some drivers that are currently consuming the protocol interface that needs to be uninstalled, so it may be dangerous to just blindly remove a protocol interface from the system. Since the usage of protocol interfaces is now being tracked for components that use the **OpenProtocol()** and **CloseProtocol()** boot services, a safe version of this function can be implemented. Before the protocol interface is removed, an attempt is made to force all the drivers that are consuming the protocol interface to stop consuming that protocol interface. This is done by looping through all the drivers that currently have the protocol interface open with an attribute of **EFI_OPEN_PROTOCOL_BY_DRIVER** or **EFI_OPEN_PROTOCOL_BY_DRIVER |**

EFI_OPEN_PROTOCOL_EXCLUSIVE and calling the boot service **DisconnectController()** for each of them. If the disconnect succeeds, then those agents will have called the boot service **CloseProtocol()** to release the protocol interface. Lastly, all of the agents that have the protocol interface open with an attribute of **EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL**, **EFI_OPEN_PROTOCOL_GET_PROTOCOL**, or **EFI_OPEN_PROTOCOL_TEST_PROTOCOL** are closed. If there are any agents remaining that still have the protocol interface open, the protocol interface is not removed from the handle and **EFI_ACCESS_DENIED** is returned. In addition, all of the drivers that were disconnected with the boot service **DisconnectController()** earlier, are reconnected with the boot service **ConnectController()**. If there are no agents remaining that are consuming the protocol interface, then the protocol interface is removed from the handle as described above.

Status Codes Returned

EFI_SUCCESS	The interface was removed.
EFI_NOT_FOUND	The interface was not found.
EFI_ACCESS_DENIED	The interface was not removed because the interface is still being used by a driver.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

ReinstallProtocolInterface()

Summary

Reinstalls a protocol interface on a device handle.

Prototype

```
typedef
EFI_STATUS
ReinstallProtocolInterface (
  IN EFI_HANDLE   Handle,
  IN EFI_GUID     *Protocol,
  IN VOID         *OldInterface,
  IN VOID         *NewInterface
  );
```

Parameters

Handle	Handle on which the interface is to be reinstalled. If *Handle* is not a valid handle, then **EFI_INVALID_PARAMETER** is returned. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description.
Protocol	The numeric ID of the interface. It is the caller's responsibility to pass in a valid GUID. See "Wired For Management Baseline" for a description of valid GUID values. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.
OldInterface	A pointer to the old interface. **NULL** can be used if a structure is not associated with *Protocol*.
NewInterface	A pointer to the new interface. **NULL** can be used if a structure is not associated with *Protocol*.

Description

The **ReinstallProtocolInterface()** function reinstalls a protocol interface on a device handle. The *OldInterface* for *Protocol* is replaced by the *NewInterface*. *NewInterface* may be the same as *OldInterface*. If it is, the registered protocol notifies occur for the handle without replacing the interface on the handle.

As with **InstallProtocolInterface()**, any process that has registered to wait for the installation of the interface is notified.

The caller is responsible for ensuring that there are no references to the *OldInterface* that is being removed.

EFI 1.10 Extension

The extension to this service directly addresses the limitations described in the section above. There may be some number of drivers currently consuming the protocol interface that is being reinstalled. In this case, it may be dangerous to replace a protocol interface in the system. It could result in an unstable state, because a driver may attempt to use the old protocol interface after a new one has been reinstalled. Since the usage of protocol interfaces is now being tracked for components that use the **OpenProtocol()** and **CloseProtocol()** boot services, a safe version of this function can be implemented.

When this function is called, a call is first made to the boot service **UninstallProtocolInterface()**. This will guarantee that all of the agents are currently consuming the protocol interface *OldInterface* will stop using *OldInterface*. If **UninstallProtocolInterface()** returns **EFI_ACCESS_DENIED**, then this function returns **EFI_ACCESS_DENIED**, *OldInterface* remains on *Handle*, and the protocol notifies are not processed because *NewInterface* was never installed.

If **UninstallProtocolInterface()** succeeds, then a call is made to the boot service **InstallProtocolInterface()** to put the *NewInterface* onto *Handle*.

Finally, the boot service **ConnectController()** is called so all agents that were forced to release *OldInterface* with **UninstallProtocolInterface()** can now consume the protocol interface *NewInterface* that was installed with **InstallProtocolInterface()**. After *OldInterface* has been replaced with *NewInterface*, any process that has registered to wait for the installation of the interface is notified.

Status Codes Returned

EFI_SUCCESS	The protocol interface was reinstalled.
EFI_NOT_FOUND	The *OldInterface* on the handle was not found.
EFI_ACCESS_DENIED	The protocol interface could not be reinstalled, because *OldInterface* is still being used by a driver that will not release it.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

RegisterProtocolNotify()

Summary

Creates an event that is to be signaled whenever an interface is installed for a specified protocol.

Prototype

```
EFI_STATUS
RegisterProtocolNotify (
      IN EFI_GUID       *Protocol,
      IN EFI_EVENT      Event,
      OUT VOID          **Registration
);
```

Parameters

Protocol	The numeric ID of the protocol for which the event is to be registered. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.
Event	Event that is to be signaled whenever a protocol interface is registered for *Protocol*. The type **EFI_EVENT** is defined in the **InstallProtocolInterface()** function description. The same **EFI_EVENT** may be used for multiple protocol notify registrations.
Registration	A pointer to a memory location to receive the registration value. This value must be saved and used by the notification function of *Event* to retrieve the list of handles that have added a protocol interface of type *Protocol*.

Description

The **RegisterProtocolNotify()** function creates an event that is to be signaled whenever a protocol interface is installed for *Protocol* by **InstallProtocolInterface()** or **ReinstallProtocolInterface()**.

Once *Event* has been signaled, the **LocateHandle()** function can be called to identify the newly installed, or reinstalled, handles that support *Protocol*. The *Registration* parameter in **RegisterProtocolNotify()** corresponds to the *SearchKey* parameter in **LocateHandle()**. Note that the same handle may be returned multiple times if the handle reinstalls the target protocol ID multiple times. This is typical for removable media devices, because when such a device reappears, it will reinstall the Block I/O protocol to indicate that the device needs to be checked again. In response, layered Disk I/O and Simple File System protocols may then reinstall their protocols to indicate that they can be re-checked, and so forth.

Status Codes Returned

EFI_SUCCESS	The notification event has been registered.
EFI_OUT_OF_RESOURCES	Space for the notification event could not be allocated.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

LocateHandle()

Summary

Returns an array of handles that support a specified protocol.

Prototype

```
EFI_STATUS
LocateHandle (
     IN EFI_LOCATE_SEARCH_TYPE   SearchType,
     IN EFI_GUID                 *Protocol OPTIONAL,
     IN VOID                     *SearchKey OPTIONAL,
     IN OUT UINTN                *BufferSize,
     OUT EFI_HANDLE              *Buffer
     );
```

Parameters

SearchType	Specifies which handle(s) are to be returned. Type **EFI_LOCATE_SEARCH_TYPE** is defined in "Related Definitions."
Protocol	Specifies the protocol to search by. This parameter is only valid if *SearchType* is **ByProtocol**. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.
SearchKey	Specifies the search key. This parameter is ignored if *SearchType* is **AllHandles** or **ByProtocol**. If *SearchType* is **ByRegisterNotify**, the parameter must be the *Registration* value returned by function **RegisterProtocolNotify()**.
BufferSize	On input, the size in bytes of *Buffer*. On output, the size in bytes of the array returned in *Buffer* (if the buffer was large enough) or the size, in bytes, of the buffer needed to obtain the array (if the buffer was not large enough).
Buffer	The buffer in which the array is returned. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description.

Related Definitions

```
//*******************************************************
// EFI_LOCATE_SEARCH_TYPE
//*******************************************************
typedef enum {
     AllHandles,
     ByRegisterNotify,
     ByProtocol
} EFI_LOCATE_SEARCH_TYPE;
```

AllHandles	*Protocol* and *SearchKey* are ignored and the function returns an array of every handle in the system.
ByRegisterNotify	*SearchKey* supplies the *Registration* value returned by **RegisterProtocolNotify()**. The function returns the next handle that is new for the registration. Only one handle is returned at a time, and the caller must loop until no more handles are returned. *Protocol* is ignored for this search type.
ByProtocol	All handles that support *Protocol* are returned. *SearchKey* is ignored for this search type.

Description

The **LocateHandle()** function returns an array of handles that match the *SearchType* request. If the input value of *BufferSize* is too small, the function returns **EFI_BUFFER_TOO_SMALL** and updates *BufferSize* to the size of the buffer needed to obtain the array.

Status Codes Returned

EFI_SUCCESS	The array of handles was returned.
EFI_NOT_FOUND	No handles match the search.
EFI_BUFFER_TOO_SMALL	The *BufferSize* is too small for the result. *BufferSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

HandleProtocol()

Summary

Queries a handle to determine if it supports a specified protocol.

Prototype

```
typedef
EFI_STATUS
HandleProtocol (
  IN  EFI_HANDLE   Handle,
  IN  EFI_GUID     *Protocol,
  OUT VOID         **Interface
  );
```

Parameters

Handle	The handle being queried. If *Handle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description.
Protocol	The published unique identifier of the protocol. It is the caller's responsibility to pass in a valid GUID. See "Wired For Management Baseline" for a description of valid GUID values. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.
Interface	Supplies the address where a pointer to the corresponding Protocol Interface is returned. **NULL** will be returned in **Interface* if a structure is not associated with *Protocol*.

Description

The **HandleProtocol()** function queries *Handle* to determine if it supports *Protocol*. If it does, then on return *Interface* points to a pointer to the corresponding Protocol Interface. *Interface* can then be passed to any protocol service to identify the context of the request.

EFI 1.10 Extension

The **HandleProtocol()** function is still available for use by old EFI applications and drivers. However, all new applications and drivers should use **OpenProtocol()** in place of **HandleProtocol()**. The following code fragment shows a possible implementation of **HandleProtocol()** using **OpenProtocol()**. The variable **EfiCoreImageHandle** is the image handle of the EFI core.

```
EFI_STATUS
HandleProtocol (
  IN  EFI_HANDLE      Handle,
  IN  EFI_GUID        *Protocol,
  OUT VOID            **Interface
  )
{
  return OpenProtocol (
           Handle,
           Protocol,
           Interface,
           EfiCoreImageHandle,
           NULL,
           EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL
           );
}
```

Status Codes Returned

EFI_SUCCESS	The interface information for the specified protocol was returned.
EFI_UNSUPPORTED	The device does not support the specified protocol.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

LocateDevicePath()

Summary

Locates the handle to a device on the device path that supports the specified protocol.

Prototype

```
EFI_STATUS
LocateDevicePath (
      IN EFI_GUID                   *Protocol,
      IN OUT EFI_DEVICE_PATH        **DevicePath,
      OUT EFI_HANDLE                *Device
      );
```

Parameters

Protocol	The protocol to search for. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.
DevicePath	On input, a pointer to a pointer to the device path. On output, the device path pointer is modified to point to the remaining part of the device path—that is, when the function finds the closest handle, it splits the device path into two parts, stripping off the front part, and returning the remaining portion. Type **EFI_DEVICE_PATH** is defined in “Related Definitions.”
Device	A pointer to the returned device handle. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description.

Related Definitions

```
//*******************************************************
// EFI_DEVICE_PATH
//*******************************************************
typedef struct _EFI_DEVICE_PATH {
UINT8       Type;
UINT8       SubType;
UINT8       Length[2];
} EFI_DEVICE_PATH;
```

Description

The **`LocateDevicePath()`** function locates all devices on *`DevicePath`* that support *`Protocol`* and returns the handle to the device that is closest to *`DevicePath`*. *`DevicePath`* is advanced over the device path nodes that were matched.

This function is useful for locating the proper instance of a protocol interface to use from a logical parent device driver. For example, a target device driver may issue the request with its own device path and locate the interfaces to perform I/O on its bus. It can also be used with a device path that contains a file path to strip off the file system portion of the device path, leaving the file path and handle to the file system driver needed to access the file.

If the handle for *`DevicePath`* supports the protocol (a direct match), the resulting device path is advanced to the device path terminator node.

Status Codes Returned

EFI_SUCCESS	The resulting handle was returned.
EFI_NOT_FOUND	No handles matched the search.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

OpenProtocol()

Summary

Queries a handle to determine if it supports a specified protocol. If the protocol is supported by the handle, it opens the protocol on behalf of the calling agent. This is an extended version of the EFI boot service **HandleProtocol()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_OPEN_PROTOCOL) (
  IN   EFI_HANDLE                    Handle,
  IN   EFI_GUID                      *Protocol,
  OUT  VOID                          **Interface       OPTIONAL,
  IN   EFI_HANDLE                    AgentHandle,
  IN   EFI_HANDLE                    ControllerHandle,
  IN   UINT32                        Attributes
  );
```

Parameters

Handle	The handle for the protocol interface that is being opened.
Protocol	The published unique identifier of the protocol. It is the callers responsibility to pass in a valid GUID. See "Wired For Management Baseline" for a description of valid GUID values.
Interface	Supplies the address where a pointer to the corresponding Protocol Interface is returned. **NULL** will be returned in **Interface* if a structure is not associated with *Protocol*. This parameter is optional, and will be ignored if *Attributes* is **EFI_OPEN_PROTOCOL_TEST_PROTOCOL**.
AgentHandle	The handle of the agent that is opening the protocol interface specified by *Protocol* and *Interface*. For agents that follow the *EFI Driver Model*, this parameter is the handle that contains the **EFI_DRIVER_BINDING_PROTOCOL** instance that is produced by the EFI Driver that is opening the protocol interface. For EFI Applications, this is the image handle of the EFI Application that is opening the protocol interface. For EFI Applications that use **HandleProtocol()** to open a protocol interface, this parameter is the image handle of the EFI firmware.

ControllerHandle	If the agent that is opening a protocol is a driver that follows the *EFI Driver Model*, then this parameter is the controller handle that requires the protocol interface. If the agent does not follow the *EFI Driver Model*, then this parameter is optional and may be **NULL**.
Attributes	The open mode of the protocol interface specified by *Handle* and *Protocol*. See "Related Definitions" for the list of legal attributes.

Description

This function opens a protocol interface on the handle specified by *Handle* for the protocol specified by *Protocol*. The first three parameters are the same as **HandleProtocol()**. The only difference is that the agent that is opening a protocol interface is tracked in EFI's internal handle database. The tracking is used by the *EFI Driver Model*, and also used to determine if it is safe to uninstall or reinstall a protocol interface.

The agent that is opening the protocol interface is specified by *AgentHandle*, *ControllerHandle*, and *Attributes*. If the protocol interface can be opened, then *AgentHandle*, *ControllerHandle*, and *Attributes* are added to the list of agents that are consuming the protocol interface specified by *Handle* and *Protocol*. In addition, the protocol interface is returned in *Interface*, and **EFI_SUCCESS** is returned. If *Attributes* is **TEST_PROTOCOL**, then *Interface* is optional, and can be **NULL**.

There are a number of reasons that this function call can return an error. If an error is returned, then *AgentHandle*, *ControllerHandle*, and *Attributes* are not added to the list of agents consuming the protocol interface specified by *Handle* and *Protocol*, and *Interface* is returned unmodified. The following is the list of conditions that must be checked before this function can return **EFI_SUCCESS**.

If *Protocol* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *Interface* is **NULL** and *Attributes* is not **TEST_PROTOCOL**, then **EFI_INVALID_PARAMETER** is returned.

If *Handle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned.

If *Handle* does not support *Protocol*, then **EFI_UNSUPPORTED** is returned.

If *Attributes* is not a legal value, then **EFI_INVALID_PARAMETER** is returned. The legal values are listed in "Related Definitions."

If *Attributes* is **BY_CHILD_CONTROLLER**, **BY_DRIVER**, **EXCLUSIVE**, or **BY_DRIVER|EXCULSIVE**, and *AgentHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned.

If *Attributes* is **BY_CHILD_CONTROLLER**, **BY_DRIVER**, or **BY_DRIVER|EXCULSIVE**, and *ControllerHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned.

If *Attributes* is **BY_CHILD_CONTROLLER** and *Handle* is identical to *ControllerHandle*, then **EFI_INVALID_PARAMETER** is returned.

If *Attributes* is **BY_DRIVER** , **BY_DRIVER|EXCLUSIVE**, or **EXCLUSIVE**, and there are any items on the open list of the protocol interface with an attribute of **EXCLUSIVE** or **BY_DRIVER|EXCLUSIVE**, then **EFI_ACCESS_DENIED** is returned.

If *Attributes* is **BY_DRIVER**, and there are any items on the open list of the protocol interface with an attribute of **BY_DRIVER**, and *AgentHandle* is the same agent handle in the open list item, then **EFI_ALREADY_STARTED** is returned.

If *Attributes* is **BY_DRIVER**, and there are any items on the open list of the protocol interface with an attribute of **BY_DRIVER**, and *AgentHandle* is different than the agent handle in the open list item, then **EFI_ACCESS_DENIED** is returned.

If *Attributes* is **BY_DRIVER|EXCLUSIVE**, and there are any items on the open list of the protocol interface with an attribute of **BY_DRIVER|EXCLUSIVE**, and *AgentHandle* is the same agent handle in the open list item, then **EFI_ALREADY_STARTED** is returned.

If *Attributes* is **BY_DRIVER|EXCLUSIVE**, and there are any items on the open list of the protocol interface with an attribute of **BY_DRIVER|EXCLUSIVE**, and *AgentHandle* is different than the agent handle in the open list item, then **EFI_ACCESS_DENIED** is returned.

If *Attributes* is **BY_DRIVER|EXCLUSIVE** or **EXCLUSIVE**, and there are any items on the open list of the protocol interface with an attribute of **BY_DRIVER**, then the boot service **DisconnectController()** is called for each of these drivers on the open list. If there are any items in the open list of the protocol interface with an attribute of **BY_DRIVER** remaining after all the **DisconnectController()** calls have been made, **EFI_ACCESS_DENIED** is returned.

Related Definitions

```
#define EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL    0x00000001
#define EFI_OPEN_PROTOCOL_GET_PROTOCOL          0x00000002
#define EFI_OPEN_PROTOCOL_TEST_PROTOCOL         0x00000004
#define EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER   0x00000008
#define EFI_OPEN_PROTOCOL_BY_DRIVER             0x00000010
#define EFI_OPEN_PROTOCOL_EXCLUSIVE             0x00000020
```

The following is the list of legal values for the *Attributes* parameter, and how each value is used.

BY_HANDLE_PROTOCOL	Used in the implementation of **HandleProtocol()**. Since **OpenProtocol()** performs the same function as **HandleProtocol()** with additional functionality, **HandleProtocol()** can simply call **OpenProtocol()** with this *Attributes* value.
GET_PROTOCOL	Used by a driver to get a protocol interface from a handle. Care must be taken when using this open mode because the driver that opens a protocol interface in this manner will not be informed if the protocol interface is uninstalled or reinstalled. The caller is also not required to close the protocol interface with **CloseProtocol()**.

TEST_PROTOCOL	Used by a driver to test for the existence of a protocol interface on a handle. *Interface* is optional for this attribute value, so it is ignored, and the caller should only use the return status code. The caller is also not required to close the protocol interface with **CloseProtocol()**.
BY_CHILD_CONTROLLER	Used by bus drivers to show that a protocol interface is being used by one of the child controllers of a bus. This information is used by the boot service **ConnectController()** to recursively connect all child controllers and by the boot service **DisconnectController()** to get the list of child controllers that a bus driver created.
BY_DRIVER	Used by a driver to gain access to a protocol interface. When this mode is used, the driver's **Stop()** function will be called by **DisconnectController()** if the protocol interface is reinstalled or uninstalled. Once a protocol interface is opened by a driver with this attribute, no other drivers will be allowed to open the same protocol interface with the **BY_DRIVER** attribute.
BY_DRIVER|EXCLUSIVE	Used by a driver to gain exclusive access to a protocol interface. If any other drivers have the protocol interface opened with an attribute of **BY_DRIVER**, then an attempt will be made to remove them with **DisconnectController()**.
EXCLUSIVE	Used by applications to gain exclusive access to a protocol interface. If any drivers have the protocol interface opened with an attribute of **BY_DRIVER**, then an attempt will be made to remove them by calling the driver's **Stop()** function.

Status Codes Returned

EFI_SUCCESS	An item was added to the open list for the protocol interface, and the protocol interface was returned in *Interface*.
EFI_INVALID_PARAMETER	*Protocol* is **NULL**.
EFI_INVALID_PARAMETER	*Interface* is **NULL**, and *Attributes* is not **TEST_PROTOCOL**.
EFI_INVALID_PARAMETER	*Handle* is not a valid **EFI_HANDLE**.
EFI_UNSUPPORTED	*Handle* does not support *Protocol*.
EFI_INVALID_PARAMETER	*Attributes* is not a legal value.
EFI_INVALID_PARAMETER	*Attributes* is **BY_CHILD_CONTROLLER** and *AgentHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Attributes* is **BY_DRIVER** and *AgentHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Attributes* is **BY_DRIVER|EXCLUSIVE** and *AgentHandle* is not a valid **EFI_HANDLE**.

continued

Status Codes Returned (continued)

EFI_INVALID_PARAMETER	*Attributes* is **EXCLUSIVE** and *AgentHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Attributes* is **BY_CHILD_CONTROLLER** and *ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Attributes* is **BY_DRIVER** and *ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Attributes* is **BY_DRIVER** | **EXCLUSIVE** and *ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Attributes* is **BY_CHILD_CONTROLLER** and *Handle* is identical to *ControllerHandle*.
EFI_ACCESS_DENIED	*Attributes* is **BY_DRIVER** and there is an item on the open list with an attribute of **BY_DRIVER** | **EXCLUSIVE** or **EXCLUSIVE**.
EFI_ACCESS_DENIED	*Attributes* is **BY_DRIVER** | **EXCLUSIVE** and there is an item on the open list with an attribute of **EXCLUSIVE**.
EFI_ACCESS_DENIED	*Attributes* is **EXCLUSIVE** and there is an item on the open list with an attribute of **BY_DRIVER** | **EXCLUSIVE** or **EXCLUSIVE**.
EFI_ALREADY_STARTED	*Attributes* is **BY_DRIVER** and there is an item on the open list with an attribute of **BY_DRIVER** whose agent handle is the same as *AgentHandle*.
EFI_ACCESS_DENIED	*Attributes* is **BY_DRIVER** and there is an item on the open list with an attribute of **BY_DRIVER** whose agent handle is different than *AgentHandle*.
EFI_ALREADY_STARTED	*Attributes* is **BY_DRIVER** | **EXCLUSIVE** and there is an item on the open list with an attribute of **BY_DRIVER** | **EXCLUSIVE** whose agent handle is the same as *AgentHandle*.
EFI_ACCESS_DENIED	*Attributes* is **BY_DRIVER** | **EXCLUSIVE** and there is an item on the open list with an attribute of **BY_DRIVER** | **EXCLUSIVE** whose agent handle is different than *AgentHandle*.
EFI_ACCESS_DENIED	*Attributes* is **BY_DRIVER** | **EXCLSUIVE** or **EXCLUSIVE** and there are items in the open list with an attribute of **BY_DRIVER** that could not be removed when **DisconnectController()** was called for that open item.

Examples

```
EFI_BOOT_SERVICES_TABLE      *gBS;
EFI_HANDLE                   ImageHandle;
EFI_DRIVER_BINDING_PROTOCOL  *This;
IN EFI_HANDLE                ControllerHandle,
extern EFI_GUID              gEfiXyzIoProtocol;
EFI_XYZ_IO_PROTOCOL          *XyzIo;
EFI_STATUS                   Status;

//
// EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL example
//   Retrieves the XYZ I/O Protocol instance from ControllerHandle
//   The application that is opening the protocol is identified by ImageHandle
//   Possible return status codes:
//     EFI_SUCCESS         : The protocol was opened and returned in XyzIo
//     EFI_UNSUPPORTED     : The protocol is not present on ControllerHandle
//
Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiXyzIoProtocol,
                  &XyzIo,
                  ImageHandle,
                  NULL,
                  EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL
                  );

//
// EFI_OPEN_PROTOCOL_GET_PROTOCOL example
//   Retrieves the XYZ I/O Protocol instance from ControllerHandle
//   The driver that is opening the protocol is identified by the
//   Driver Binding Protocol instance This.  This->DriverBindingHandle
//   identifies the agent that is opening the protocol interface, and it
//   is opening this protocol on behalf of ControllerHandle.
//   Possible return status codes:
//     EFI_SUCCESS         : The protocol was opened and returned in XyzIo
//     EFI_UNSUPPORTED     : The protocol is not present on ControllerHandle
//
Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiXyzIoProtocol,
                  &XyzIo,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );

//
// EFI_OPEN_PROTOCOL_TEST_PROTOCOL example
//   Tests to see if the XYZ I/O Protocol is present on ControllerHandle
//   The driver that is opening the protocol is identified by the
//   Driver Binding Protocol instance This.  This->DriverBindingHandle
//   identifies the agent that is opening the protocol interface, and it
//   is opening this protocol on behalf of ControllerHandle.
//     EFI_SUCCESS         : The protocol was opened and returned in XyzIo
//     EFI_UNSUPPORTED     : The protocol is not present on ControllerHandle
//
Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiXyzIoProtocol,
                  NULL,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_TEST_PROTOCOL
                  );

//
// EFI_OPEN_PROTOCOL_BY_DRIVER example
//   Opens the XYZ I/O Protocol on ControllerHandle
//   The driver that is opening the protocol is identified by the
//   Driver Binding Protocol instance This.  This->DriverBindingHandle
//   identifies the agent that is opening the protocol interface, and it
//   is opening this protocol on behalf of ControllerHandle.
//   Possible return status codes:
//     EFI_SUCCESS         : The protocol was opened and returned in XyzIo
//     EFI_UNSUPPORTED     : The protocol is not present on ControllerHandle
//     EFI_ALREADY_STARTED : The protocol is already opened by the driver
//     EFI_ACCESS_DENIED   : The protocol is managed by a different driver
//
Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiXyzIoProtocol,
                  &XyzIo,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_BY_DRIVER
                  );

//
// EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE example
//   Opens the XYZ I/O Protocol on ControllerHandle
//   The driver that is opening the protocol is identified by the
//   Driver Binding Protocol instance This.  This->DriverBindingHandle
//   identifies the agent that is opening the protocol interface, and it
//   is opening this protocol on behalf of ControllerHandle.
//   Possible return status codes:
//     EFI_SUCCESS         : The protocol was opened and returned in XyzIo.  If
//                           a different driver had the XYZ I/O Protocol opened
//                           BY_DRIVER, then that driver was disconnected to
//                           allow this driver to open the XYZ I/O Protocol.
//     EFI_UNSUPPORTED     : The protocol is not present on ControllerHandle
//     EFI_ALREADY_STARTED : The protocol is already opened by the driver
//     EFI_ACCESS_DENIED   : The protocol is managed by a different driver that
//                           already has the protocol opened with an EXCLUSIVE
//                           attribute.
//
Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiXyzIoProtocol,
                  &XyzIo,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE
                  );
```

CloseProtocol()

Summary

Closes a protocol on a handle that was opened using **OpenProtocol()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CLOSE_PROTOCOL) (
  IN EFI_HANDLE                   Handle,
  IN EFI_GUID                     *Protocol,
  IN EFI_HANDLE                   AgentHandle,
  IN EFI_HANDLE                   ControllerHandle
  );
```

Parameters

Handle	The handle for the protocol interface that was previously opened with **OpenProtocol()**, and is now being closed.
Protocol	The published unique identifier of the protocol. It is the caller's responsibility to pass in a valid GUID. See "Wired For Management Baseline" for a description of valid GUID values.
AgentHandle	The handle of the agent that is closing the protocol interface. For agents that follow the *EFI Driver Model*, this parameter is the handle that contains the **EFI_DRIVER_BINDING_PROTOCOL** instance that is produced by the EFI Driver that is opening the protocol interface. For EFI Applications, this is the image handle of the EFI Application. For EFI Applications that used **HandleProtocol()** to open the protocol interface, this will be the image handle of the EFI firmware.
ControllerHandle	If the agent that opened a protocol is a driver that follows the *EFI Driver Model*, then this parameter is the controller handle that required the protocol interface. If the agent does not follow the *EFI Driver Model*, then this parameter is optional and may be **NULL**.

Description

This function updates the handle database to show that the protocol instance specified by *Handle* and *Protocol* is no longer required by the agent and controller specified *AgentHandle* and *ControllerHandle*.

If *Handle* or *AgentHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. If *ControllerHandle* is not **NULL**, and *ControllerHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. If *Protocol* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If the interface specified by *Protocol* is not supported by the handle specified by *Handle*, then **EFI_NOT_FOUND** is returned.

If the interface specified by *Protocol* is supported by the handle specified by *Handle*, then a check is made to see if the protocol instance specified by *Protocol* and *Handle* was opened by *AgentHandle* and *ControllerHandle* with **OpenProtocol()**. If the protocol instance was not opened by *AgentHandle* and *ControllerHandle*, then **EFI_NOT_FOUND** is returned. If the protocol instance was opened by *AgentHandle* and *ControllerHandle*, then all of those references are removed from the handle database, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The protocol instance was closed.
EFI_INVALID_PARAMETER	*Handle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*AgentHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ControllerHandle* is not **NULL** and *ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Protocol* is **NULL**.
EFI_NOT_FOUND	*Handle* does not support the protocol specified by *Protocol*.
EFI_NOT_FOUND	The protocol interface specified by *Handle* and *Protocol* is not currently open by *AgentHandle* and *ControllerHandle*.

Examples

```
EFI_BOOT_SERVICES_TABLE      *gBS;
EFI_HANDLE                   ImageHandle;
EFI_DRIVER_BINDING_PROTOCOL  *This;
IN EFI_HANDLE                ControllerHandle,
extern EFI_GUID              gEfiXyzIoProtocol;
EFI_STATUS                   Status;

//
// Close the XYZ I/O Protocol that was opened on behalf of ControllerHandle
//
Status = gBS->CloseProtocol (
                ControllerHandle,
                &gEfiXyzIoProtocol,
                This->DriverBindingHandle,
                ControllerHandle
                );

//
// Close the XYZ I/O Protocol that was opened with BY_HANDLE_PROTOCOL
//
Status = gBS->CloseProtocol (
                ControllerHandle,
                &gEfiXyzIoProtocol,
                ImageHandle,
                NULL
                );
```

OpenProtocolInformation()

Summary

Retrieves the list of agents that currently have a protocol interface opened.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_OPEN_PROTOCOL_INFORMATION) (
  IN  EFI_HANDLE                          Handle,
  IN  EFI_GUID                            *Protocol,
  OUT EFI_OPEN_PROTOCOL_INFORMATION_ENTRY **EntryBuffer,
  OUT UINTN                               *EntryCount
  );
```

Parameters

Handle	The handle for the protocol interface that is being queried.
Protocol	The published unique identifier of the protocol. It is the callers responsibility to pass in a valid GUID. See “Wired For Management Baseline” for a description of valid GUID values.
EntryBuffer	A pointer to a buffer of open protocol information in the form of **EFI_OPEN_PROTOCOL_INFORMATION_ENTRY** structures. See "Related Definitions" for the declaration of this type. The buffer is allocated by this service, and it is the caller's responsibility to free this buffer when the caller no longer requires the buffer's contents.
EntryCount	A pointer to the number of entries in *EntryBuffer*.

Related Definitions

```
typedef struct {
  EFI_HANDLE                AgentHandle;
  EFI_HANDLE                ControllerHandle;
  UINT32                    Attributes;
  UINT32                    OpenCount;
} EFI_OPEN_PROTOCOL_INFORMATION_ENTRY;
```

Description

This function allocates and returns a buffer of **EFI_OPEN_PROTOCOL_INFORMATION_ENTRY** structures. The buffer is returned in *EntryBuffer*, and the number of entries is returned in *EntryCount*.

If the interface specified by *Protocol* is not supported by the handle specified by *Handle*, then **EFI_NOT_FOUND** is returned.

If the interface specified by *Protocol* is supported by the handle specified by *Handle*, then *EntryBuffer* is allocated with the boot service **AllocatePool()**, and *EntryCount* is set to the number of entries in *EntryBuffer*. Each entry of *EntryBuffer* is filled in with the image handle, controller handle, and attributes that were passed to **OpenProtocol()** when the protocol interface was opened. The field **OpenCount** shows the number of times that the protocol interface has been opened by the agent specified by **ImageHandle**, **ControllerHandle**, and **Attributes**. After the contents of *EntryBuffer* have been filled in, **EFI_SUCCESS** is returned. It is the caller's responsibility to call **FreePool()** on *EntryBuffer* when the caller no longer required the contents of *EntryBuffer*.

If there are not enough resources available to allocate *EntryBuffer*, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The open protocol information was returned in *EntryBuffer*, and the number of entries was returned *EntryCount*.
EFI_NOT_FOUND	*Handle* does not support the protocol specified by *Protocol*.
EFI_OUT_OF_RESOURCES	There are not enough resources available to allocate *EntryBuffer*.

Examples

See example in the **LocateHandleBuffer()** function description for an example on how **LocateHandleBuffer()**, **ProtocolsPerHandle()**, **OpenProtocol()**, and **OpenProtocolInformation()** can be used to traverse the entire handle database.

ConnectController()

Summary

Connects one or more drivers to a controller.

Prototype

```
typedef
EFI_STATUS
ConnectController (
  IN  EFI_HANDLE                ControllerHandle,
  IN  EFI_HANDLE                *DriverImageHandle     OPTIONAL,
  IN  EFI_DEVICE_PATH_PROTOCOL  *RemainingDevicePath   OPTIONAL,
  IN  BOOLEAN                   Recursive
  );
```

Parameters

ControllerHandle	The handle of the controller to which driver(s) are to be connected.
DriverImageHandle	A pointer to an ordered list of driver image handles. The list is terminated by a **NULL** image handle. These driver image handles are candidates for the driver(s) that will manage the controller specified by *ControllerHandle*. This is an optional parameter that may be **NULL**. This parameter is typically used to debug new drivers.
RemainingDevicePath	A pointer to the device path that specifies a child of the controller specified by *ControllerHandle*. This is an optional parameter that may be **NULL**. If it is **NULL**, then handles for all the children of *ControllerHandle* will be created. This parameter is passed unchanged to the **Supported()** and **Start()** services of the **EFI_DRIVER_BINDING_PROTOCOL** attached to *ControllerHandle*.
Recursive	If **TRUE**, then **ConnectController()** is called recursively until the entire tree of controllers below the controller specified by *ControllerHandle* have been created. If **FALSE**, then the tree of controllers is only expanded one level.

Description

This function connects one or more drivers to the controller specified by *ControllerHandle*. If *ControllerHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. If there are no **EFI_DRIVER_BINDING_PROTOCOL** instances present in the system, then return **EFI_NOT_FOUND**. If there are not enough resources available to complete this function, then **EFI_OUT_OF_RESOURCES** is returned.

If *Recursive* is **FALSE**, then this function returns after all drivers have been connected to *ControllerHandle*. If *Recursive* is **TRUE**, then **ConnectController()** is called recursively on all of the child controllers of *ControllerHandle*. The child controllers can be identified by searching the handle database for all the controllers that have opened *ControllerHandle* with an attribute of **EFI_OPEN_PROTOCOL_BY_CHILD_ CONTROLLER**.

This functions uses four precedence rules when deciding the order that drivers are tested against controllers. These four rules from highest precedence to lowest precedence are as follows:

1. ***Context Override*** : *DriverImageHandle* is an ordered list of image handles. The highest priority image handle is the first element of the list, and the lowest priority image handle is the last element of the list. The list is terminated with a **NULL** image handle.
2. ***Platform Driver Override*** : If an **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL** instance is present in the system, then the **GetDriver()** service of this protocol is used to retrieve an ordered list of image handles for *ControllerHandle*. The first image handle returned from **GetDriver()** has the highest precedence, and the last image handle returned from **GetDriver()** has the lowest precedence. The ordered list is terminated when **GetDriver()** returns **EFI_NOT_FOUND**. It is legal for no image handles to be returned by **GetDriver()**. There can be at most a single instance in the system of the **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL**. If there is more than one, then the system behavior is not deterministic.
3. ***Bus Specific Driver Override*** : If there is an instance of the **EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL** attached to *ControllerHandle*, then the **GetDriver()** service of this protocol is used to retrieve an ordered list of image handle for *ControllerHandle*. The first image handle returned from **GetDriver()** has the highest precedence, and the last image handle returned from **GetDriver()** has the lowest precedence. The ordered list is terminated when **GetDriver()** returns **EFI_NOT_FOUND**. It is legal for no image handles to be returned by **GetDriver()**.
4. ***Driver Binding Search*** : The list of available driver image handles can be found by using the boot service **LocateHandle()** with a *SearchType* of *ByProtocol* for the GUID of the **EFI_DRIVER_BINDING_PROTOCOL**. From this list, the image handles found in rules (1), (2), and (3) above are removed. The remaining image handles are sorted from highest to lowest based on the *Version* field of the **EFI_DRIVER_BINDING_PROTOCOL** instance associated with each image handle.

Each of the four groups of image handles listed above is tested against *ControllerHandle* in order by using the **EFI_DRIVER_BINDING_PROTOCOL** service **Supported()**. *RemainingDevicePath* is passed into **Supported()** unmodified. The first image handle whose **Supported()** service returns **EFI_SUCCESS** is marked so the image handle will not be tried again during this call to **ConnectController()**. Then, the **Start()** service of the **EFI_DRIVER_BINDING_PROTOCOL** is called for *ControllerHandle*. Once again, *RemainingDevicePath* is passed in unmodified. Every time **Supported()** returns **EFI_SUCCESS**, the search for drivers restarts with the highest precedence image handle. This process is repeated until no image handles pass the **Supported()** check.

If at least one image handle returned **EFI_SUCCESS** from its **Start()** service, then **EFI_SUCCESS** is returned.

If no image handles returned **EFI_SUCCESS** from their **Start()** service then **EFI_NOT_FOUND** is returned unless *RemainingDevicePath* is not **NULL**, and *RemainingDevicePath* is an End Node. In this special case, **EFI_SUCCESS** is returned because it is not an error to fail to start a child controller that is specified by an End Device Path Node.

Status Codes Returned

EFI_SUCCESS	One or more drivers were connected to *ControllerHandle*.
EFI_SUCCESS	No drivers were connected to *ControllerHandle*, but *RemainingDevicePath* is not **NULL**, and it is an End Device Path Node.
EFI_INVALID_PARAMETER	*ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_NOT_FOUND	There are no **EFI_DRIVER_BINDING_PROTOCOL** instances present in the system.
EFI_NOT_FOUND	No drivers were connected to *ControllerHandle*.

Examples

```
//
// Connect All Handles Example
//   The following example recusively connects all controllers in a platform.
//

EFI_STATUS                            Status;
EFI_BOOT_SERVICES_TABLE               *gBS;
UINTN                                 HandleCount;
EFI_HANDLE                            *HandleBuffer;
UINTN                                 HandleIndex;

//
// Retrieve the list of all handles from the handle database
//
Status = gBS->LocateHandleBuffer (
                AllHandles,
                NULL,
                NULL,
                &HandleCount,
                &HandleBuffer
                );
if (!EFI_ERROR (Status)) {
  for (HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++) {
    Status = gBS->ConnectController (
                    HandleBuffer[HandleIndex],
                    NULL,
                    NULL,
                    TRUE
                    );
  }
  gBS->FreePool(HandleBuffer);
}

//
// Connect Device Path Example
//   The following example walks the device path nodes of a device path, and
//   connects only the drivers required to force a handle with that device path
//   to be present in the handle database.  This algorithms guarantees that
//   only the minimum number of devices and drivers are initialized.
//

EFI_STATUS                  Status;
EFI_DEVICE_PATH_PROTOCOL    *DevicePath;
EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath;
EFI_HANDLE                  Handle;
```

```
do {
  //
  // Find the handle that best matches the Device Path. If it is only a
  // partial match the remaining part of the device path is returned in
  // RemainingDevicePath.
  //
  RemainingDevicePath = DevicePath;
  Status = gBS->LocateDevicePath (
                  &gEfiDevicePathProtocolGuid,
                  &RemainingDevicePath,
                  &Handle
                  );
  if (EFI_ERROR(Status)) {
    return EFI_NOT_FOUND;
  }

  //
  // Connect all drivers that apply to Handle and RemainingDevicePath
  // If no drivers are connected Handle, then return EFI_NOT_FOUND
  // The Recursive flag is FALSE so only one level will be expanded.
  //
  Status = gBS->ConnectController (
                  Handle,
                  NULL,
                  RemainingDevicePath,
                  FALSE
                  );
  if (EFI_ERROR(Status)) {
    return EFI_NOT_FOUND;
  }

  //
  // Loop until RemainingDevicePath is an empty device path
  //
} while (!IsDevicePathEnd (RemainingDevicePath));

//
// A handle with DevicePath exists in the handle database
//
return EFI_SUCCESS;
```

DisconnectController()

Summary

Disconnects one or more drivers from a controller.

Prototype

```
typedef
EFI_STATUS
DisconnectController (
  IN  EFI_HANDLE  ControllerHandle,
  IN  EFI_HANDLE  DriverImageHandle  OPTIONAL,
  IN  EFI_HANDLE  ChildHandle        OPTIONAL
  );
```

Parameters

ControllerHandle	The handle of the controller from which driver(s) are to be disconnected.
DriverImageHandle	The driver to disconnect from *ControllerHandle*. If *DriverImageHandle* is **NULL**, then all the drivers currently managing *ControllerHandle* are disconnected from *ControllerHandle*.
ChildHandle	The handle of the child to destroy. If *ChildHandle* is **NULL**, then all the children of *ControllerHandle* are destroyed before the drivers are disconnected from *ControllerHandle*.

Description

This function disconnects one or more drivers from the controller specified by *ControllerHandle*. If *DriverImageHandle* is **NULL**, then all of the drivers currently managing *ControllerHandle* are disconnected from *ControllerHandle*. If *DriverImageHandle* is not **NULL**, then only the driver specified by *DriverImageHandle* is disconnected from *ControllerHandle*. If *ChildHandle* is **NULL**, then all of the children of *ControllerHandle* are destroyed before the drivers are disconnected from *ControllerHandle*. If *ChildHandle* is not **NULL**, then only the child controller specified by *ChildHandle* is destroyed. If *ChildHandle* is the only child of *ControllerHandle*, then the driver specified by *DriverImageHandle* will be disconnected from *ControllerHandle*. A driver is disconnected from a controller by calling the **Stop()** service of the **EFI_DRIVER_BINDING_PROTOCOL**. The **EFI_DRIVER_BINDING_PROTOCOL** is on the driver image handle, and the handle of the controller is passed into the **Stop()** service. The list of drivers managing a controller, and the list of children for a specific controller can be retrieved from the handle database with the boot service **OpenProtocolInformation()**. If all the required drivers are disconnected from *ControllerHandle*, then **EFI_SUCCESS** is returned.

If *ControllerHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. If no drivers are managing *ControllerHandle*, then **EFI_SUCCESS** is returned. If *DriverImageHandle* is not **NULL**, and *DriverImageHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. If *DriverImageHandle* is not **NULL**, and *DriverImageHandle* is not currently managing *ControllerHandle*, then **EFI_SUCCESS** is returned. If *ChildHandle* is not **NULL**, and *ChildHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. If there are not enough resources available to disconnect drivers from *ControllerHandle*, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	One or more drivers were disconnected from the controller.
EFI_SUCCESS	On entry, no drivers are managing *ControllerHandle*.
EFI_SUCCESS	*DriverImageHandle* is not **NULL**, and on entry *DriverImageHandle* is not managing *ControllerHandle*.
EFI_INVALID_PARAMETER	*ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*DriverImageHandle* is not **NULL**, and it is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ChildHandle* is not **NULL**, and it is not a valid **EFI_HANDLE**.
EFI_OUT_OF_RESOURCES	There are not enough resources available to disconnect any drivers from *ControllerHandle*.
EFI_DEVICE_ERROR	The controller could not be disconnected because of a device error.

Examples

```
//
// Disconnect All Handles Example
//   The following example recusively disconnects all drivers from all
//   controllers in a platform.
//

EFI_STATUS                          Status;
EFI_BOOT_SERVICES_TABLE             *gBS;
UINTN                               HandleCount;
EFI_HANDLE                          *HandleBuffer;
UINTN                               HandleIndex;

//
// Retrieve the list of all handles from the handle database
//
Status = gBS->LocateHandleBuffer (
                AllHandles,
                NULL,
                NULL,
                &HandleCount,
                &HandleBuffer
                );
if (!EFI_ERROR (Status)) {
  for (HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++) {
    Status = gBS->DisconnectController (
                    HandleBuffer[HandleIndex],
                    NULL,
                    NULL
                    );
  }
  gBS->FreePool(HandleBuffer);
```

ProtocolsPerHandle()

Summary

Retrieves the list of protocol interface GUIDs that are installed on a handle in a buffer allocated from pool.

Prototype

```
typedef
EFI_STATUS
ProtocolsPerHandle (
  IN  EFI_HANDLE  Handle,
  OUT EFI_GUID    ***ProtocolBuffer,
  OUT UINTN       *ProtocolBufferCount
  );
```

Parameters

Handle	The handle from which to retrieve the list of protocol interface GUIDs.
ProtocolBuffer	A pointer to the list of protocol interface GUID pointers that are installed on *Handle*. This buffer is allocated with a call to the Boot Service **AllocatePool()**. It is the caller's responsibility to call the Boot Service **FreePool()** when the caller no longer requires the contents of *ProtocolBuffer*.
ProtocolBufferCount	A pointer to the number of GUID pointers present in *ProtocolBuffer*.

Description

The **ProtocolsPerHandle()** function retrieves the list of protocol interface GUIDs that are installed on *Handle*. The list is returned in *ProtocolBuffer*, and the number of GUID pointers in *ProtocolBuffer* is returned in *ProtocolBufferCount*.

If *Handle* is **NULL** or *Handle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned.

If *ProtocolBuffer* is **NULL**, then **EFI_INVALID_PAREMETER** is returned.

If *ProtocolBufferCount* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If there are not enough resources available to allocate *ProtocolBuffer*, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The list of protocol interface GUIDs installed on *Handle* was returned in *ProtocolBuffer*. The number of protocol interface GUIDs was returned in *ProtocolBufferCount*.
EFI_INVALID_PARAMETER	*Handle* is **NULL**.
EFI_INVALID_PARAMETER	*Handle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ProtocolBuffer* is **NULL**.
EFI_INVALID_PARAMETER	*ProtocolBufferCount* is **NULL**.
EFI_OUT_OF_RESOURCES	There is not enough pool memory to store the results.

Examples

See example in the **LocateHandleBuffer()** function description for an example on how **LocateHandleBuffer()**, **ProtocolsPerHandle()**, **OpenProtocol()**, and **OpenProtocolInformation()** can be used to traverse the entire handle database.

LocateHandleBuffer()

Summary

Returns an array of handles that support the requested protocol in a buffer allocated from pool.

Prototype

```
typedef
EFI_STATUS
LocateHandleBuffer (
  IN EFI_LOCATE_SEARCH_TYPE  SearchType,
  IN EFI_GUID                *Protocol   OPTIONAL,
  IN VOID                    *SearchKey  OPTIONAL,
  IN OUT UINTN               *NoHandles,
  OUT EFI_HANDLE             **Buffer
  );
```

Parameters

SearchType	Specifies which handle(s) are to be returned.
Protocol	Provides the protocol to search by. This parameter is only valid for a *SearchType* of **ByProtocol**.
SearchKey	Supplies the search key depending on the *SearchType*.
NoHandles	The number of handles returned in *Buffer*.
Buffer	A pointer to the buffer to return the requested array of handles that support *Protocol*. This buffer is allocated with a call to the Boot Service **AllocatePool()**. It is the caller's responsibility to call the Boot Service **FreePool()** when the caller no longer requires the contents of *Buffer*.

Description

The **LocateHandleBuffer()** function returns one or more handles that match the *SearchType* request. *Buffer* is allocated from pool, and the number of entries in *Buffer* is returned in *NoHandles*. Each *SearchType* is described below:

AllHandles	*Protocol* and *SearchKey* are ignored and the function returns an array of every handle in the system.
ByRegisterNotify	*SearchKey* supplies the Registration returned by **RegisterProtocolNotify()**. The function returns the next handle that is new for the Registration. Only one handle is returned at a time, and the caller must loop until no more handles are returned. *Protocol* is ignored for this search type.
ByProtocol	All handles that support *Protocol* are returned. *SearchKey* is ignored for this search type.

If *NoHandles* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *Buffer* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If there are no handles in the handle database that match the search criteria, then **EFI_NOT_FOUND** is returned.

If there are not enough resources available to allocate *Buffer*, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The array of handles was returned in *Buffer*, and the number of handles in *Buffer* was returned in *NoHandles*.
EFI_INVALID_PARAMETER	*NoHandles* is **NULL**
EFI_INVALID_PARAMETER	*Buffer* is **NULL**
EFI_NOT_FOUND	No handles match the search.
EFI_OUT_OF_RESOURCES	There is not enough pool memory to store the matching results.

Examples

```
//
// The following example traverses the entire handle database.  First all of
// the handles in the handle database are retrieved by using
// LocateHandleBuffer().  Then it uses ProtocolsPerHandle() to retrieve the
// list of protocol GUIDs attached to each handle.  Then it uses OpenProtocol()
// to get the protocol instance associated with each protocol GUID on the
// handle.  Finally, it uses OpenProtocolInformation() to retrieve the list of
// agents that have opened the protocol on the handle.  The caller of these
// functions must make sure that they free the return buffers with FreePool()
// when they are done.
//

EFI_STATUS                          Status;
EFI_BOOT_SERVICES_TABLE             *gBS;
EFI_HANDLE                          ImageHandle;
UINTN                               HandleCount;
EFI_HANDLE                          *HandleBuffer;
UINTN                               HandleIndex;
EFI_GUID                            **ProtocolGuidArray;
UINTN                               ArrayCount;
UINTN                               ProtocolIndex;
EFI_OPEN_PROTOCOL_INFORMATION_ENTRY *OpenInfo;
UINTN                               OpenInfoCount;
UINTN                               OpenInfoIndex;

//
// Retrieve the list of all handles from the handle database
//
Status = gBS->LocateHandleBuffer (
                AllHandles,
                NULL,
                NULL,
                &HandleCount,
                &HandleBuffer
                );
if (!EFI_ERROR (Status)) {
  for (HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++) {
      //
      // Retrieve the list of all the protocols on each handle
      //
      Status = gBS->ProtocolsPerHandle (
                      HandleBuffer[HandleIndex],
                      &ProtocolGuidArray,
                      &ArrayCount
                      );
      if (!EFI_ERROR (Status)) {
        for (ProtocolIndex = 0; ProtocolIndex < ArrayCount; ProtocolIndex++) {
          //
          // Retrieve the protocol instance for each protocol
          //
          Status = gBS->OpenProtocol (
                          HandleBuffer[HandleIndex],
                          ProtocolGuidArray[ProtocolIndex],
                          &Instance,
                          ImageHandle,
                          NULL,
                          EFI_OPEN_PROTOCOL_GET_PROTOCOL
                          );

          //
          // Retrieve the list of agents that have opened each protocol
          //
          Status = gBS->OpenProtocolInformation (
                          HandleBuffer[HandleIndex],
                          ProtocolGuidArray[ProtocolIndex],
                          &OpenInfo,
                          &OpenInfoCount
                          );
          if (!EFI_ERROR (Status)) {
            for (OpenInfoIndex=0;OpenInfoIndex<OpenInfoCount;OpenInfoIndex++) {
              //
              // HandleBuffer[HandleIndex] is the handle
              // ProtocolGuidArray[ProtocolIndex] is the protocol GUID
              // Instance is the protocol instance for the protocol
              // OpenInfo[OpenInfoIndex] is an agent that has opened a protocol
              //
            }
            if (OpenInfo != NULL) {
              gBS->FreePool(OpenInfo);
            }
          }
        }
        if (ProtocolGuidArray != NULL) {
          gBS->FreePool(ProtocolGuidArray);
        }
      }
    }
    if (HandleBuffer != NULL) {
      gBS->FreePool (HandleBuffer);
    }
  }
```

LocateProtocol()

Summary

Returns the first protocol instance that matches the given protocol.

Prototype

```
typedef
EFI_STATUS
LocateProtocol (
  IN  EFI_GUID   *Protocol,
  IN  VOID       *Registration   OPTIONAL,
  OUT VOID       **Interface
  );
```

Parameters

Protocol	Provides the protocol to search for.
Registration	Optional registration key returned from **RegisterProtocolNotify()**. If *Registration* is **NULL**, then it is ignored.
Interface	On return, a pointer to the first interface that matches *Protocol* and *Registration*.

Description

The **LocateProtocol()** function finds the first device handle that support *Protocol*, and returns a pointer to the protocol interface from that handle in *Interface*. If no protocol instances are found, then *Interface* is set to **NULL**.

If *Interface* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *Registration* is **NULL**, and there are no handles in the handle database that support *Protocol*, then **EFI_NOT_FOUND** is returned.

If *Registration* is not **NULL**, and there are no new handles for *Registration*, then **EFI_NOT_FOUND** is returned.

Status Codes Returned

EFI_SUCCESS	A protocol instance matching *Protocol* was found and returned in *Interface*.
EFI_INVALID_PARAMETER	*Interface* is **NULL**.
EFI_NOT_FOUND	No protocol instances were found that match *Protocol*.

InstallMultipleProtocolInterfaces()

Summary

Installs one or more protocol interfaces into the boot services environment.

Prototype

```
typedef
EFI_STATUS
InstallMultipleProtocolInterfaces (
  IN OUT EFI_HANDLE   *Handle,
  ...
  );
```

Parameters

Handle	The handle to install the new protocol interfaces on, or **NULL** if a new handle is to be allocated.
...	A variable argument list containing pairs of protocol GUIDs and protocol interfaces.

Description

This function installs a set of protocol interfaces into the boot services environment. It removes arguments from the variable argument list in pairs. The first item is always a pointer to the protocol's GUID, and the second item is always a pointer to the protocol's interface. These pairs are used to call the boot service **InstallProtocolInterface()** to add a protocol interface to *Handle*. If *Handle* is **NULL** on entry, then a new handle will be allocated. The pairs of arguments are removed in order from the variable argument list until a **NULL** protocol GUID value is found. If any errors are generated while the protocol interfaces are being installed, then all the protocols installed prior to the error will be uninstalled with the boot service **UninstallProtocolInterface()** before the error is returned. The same GUID cannot be installed more than once onto the same handle. If the same GUID is installed more than once onto the same handle, then the results are not predictable.

It is illegal to have two handles in the handle database with identical device paths. This service performs a test to guarantee a duplicate device path is not inadvertently installed on two different handles. Before any protocol interfaces are installed onto *Handle*, the list of GUID/pointer pair parameters are searched to see if a Device Path Protocol instance is being installed. If a Device Path Protocol instance is going to be installed onto *Handle*, then a check is made to see if a handle is already present in the handle database with an identical Device Path Protocol instance. If an identical Device Path Protocol instance is already present in the handle database, then no protocols are installed onto *Handle*, and **EFI_ALREADY_STARTED** is returned.

Status Codes Returned

EFI_SUCCESS	All the protocol interfaces were installed.
EFI_ALREADY_STARTED	A Device Path Protocol instance was passed in that is already present in the handle database.
EFI_OUT_OF_RESOURCES	There was not enough memory in pool to install all the protocols.

UninstallMultipleProtocolInterfaces()

Summary

Removes one or more protocol interfaces into the boot services environment.

Prototype

```
typedef
EFI_STATUS
UninstallMultipleProtocolInterfaces (
  IN EFI_HANDLE  Handle,
  ...
  );
```

Parameters

Handle	The handle to remove the protocol interfaces from.
...	A variable argument list containing pairs of protocol GUIDs and protocol interfaces.

Description

This function removes a set of protocol interfaces from the boot services environment. It removes arguments from the variable argument list in pairs. The first item is always a pointer to the protocol's GUID, and the second item is always a pointer to the protocol's interface. These pairs are used to call the boot service **UninstallProtocolInterface()** to remove a protocol interface from *Handle*. The pairs of arguments are removed in order from the variable argument list until a **NULL** protocol GUID value is found. If all of the protocols are uninstalled from *Handle*, then **EFI_SUCCESS** is returned. If any errors are generated while the protocol interfaces are being uninstalled, then the protocols uninstalled prior to the error will be reinstalled with the boot service **InstallProtocolInterface()** and the status code **EFI_INVALID_PARAMETER** is returned.

Status Codes Returned

EFI_SUCCESS	All the protocol interfaces were removed.
EFI_INVALID_PARAMETER	One of the protocol interfaces was not previously installed on *Handle*.

5.4 Image Services

Three types of images can be loaded: EFI Applications, EFI Boot Services Drivers, and EFI Runtime Services Drivers. An EFI OS Loader is a type of EFI Application. The most significant difference between these image types is the type of memory into which they are loaded by the firmware's loader. Table 5-8 summarizes the differences between images.

Table 5-8. Image Type Differences Summary

	EFI Application	EFI Boot Services Driver	EFI Runtime Services Driver
Description	A transient application that is loaded during boot services time. EFI applications are either unloaded when they complete, or they take responsibility for the continued operation of the system via `ExitBootServices().` The applications are loaded in sequential order by the boot manager, but one application may dynamically load another.	A program that is loaded into boot services memory and stays resident until boot services terminates.	A program that is loaded into runtime services memory and stays resident during runtime. The memory required for a Runtime Services Driver must be performed in a single memory allocation, and marked as `EfiRuntimeServicesData`. (Note that the memory only stays resident when booting an EFI-compatible operating system. Legacy operating systems will reuse the memory.)
Loaded into memory type	`EfiLoaderCode`, `EfiLoaderData`	`EfiBootServicesCode`, `EfiBootServicesData`	`EfiRuntimeServicesCode,` `EfiRuntimeServicesData`
Default pool allocations from memory type	`EfiLoaderData`	`EfiBootServicesData`	`EfiRuntimeServicesData`
Exit behavior	When an application exits, firmware frees the memory used to hold its image.	When a boot services driver exits with an error code, firmware frees the memory used to hold its image. When a boot services driver's entry point completes with `EFI_SUCCESS,` the image is retained in memory.	When a runtime services driver exits with an error code, firmware frees the memory used to hold its image. When a runtime services driver's entry point completes with `EFI_SUCCESS`, the image is retained in memory.
Notes	This type of image would not install any protocol interfaces or handles.	This type of image would typically use `InstallProtocolInterface()`.	A runtime driver can only allocate runtime memory during boot services time. Due to the complexity of performing a virtual relocation for a runtime image, this driver type is discouraged unless it is absolutely required.

Most images are loaded by the boot manager. When an EFI application or driver is installed, the installation procedure registers itself with the boot manager for loading. However, in some cases an application or driver may want to programmatically load and start another EFI image. This can be done with the `LoadImage()` and `StartImage()` interfaces. Drivers may only load applications during the driver's initialization entry point. Table 5-9 lists the functions that make up Image Services.

Table 5-9. Image Functions

Name	Type	Description
LoadImage	Boot	Loads an EFI image into memory.
StartImage	Boot	Transfers control to a loaded image's entry point.
UnloadImage	Boot	Unloads an image.
EFI_IMAGE_ENTRY_POINT	Boot	Prototype of an EFI Image's entry point.
Exit	Boot	Exits the image's entry point.
ExitBootServices	Boot	Terminates boot services.

The Image boot services have been modified to take advantage of the information that is now being tracked with the `OpenProtocol()` and `CloseProtocol()` boot services. Since the usage of protocol interfaces is being tracked with these new boot services, it is now possible to automatically close protocol interfaces when an EFI Application or an EFI Driver is unloaded or exited.

LoadImage()

Summary

Loads an EFI image into memory.

Prototype

```
EFI_STATUS
LoadImage (
    IN BOOLEAN                          BootPolicy,
    IN EFI_HANDLE                       ParentImageHandle,
    IN EFI_DEVICE_PATH                  *FilePath,
    IN VOID                             *SourceBuffer OPTIONAL,
    IN UINTN                            SourceSize,
    OUT EFI_HANDLE                      *ImageHandle
    );
```

Parameters

BootPolicy	If **TRUE**, indicates that the request originates from the boot manager, and that the boot manager is attempting to load *FilePath* as a boot selection. Ignored if *SourceBuffer* is not **NULL**.
ParentImageHandle	The caller's image handle. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description. This field is used to initialize the *ParentHandle* field of the **EFI_LOADED_IMAGE** protocol for the image that is being loaded.
FilePath	The *DeviceHandle* specific file path from which the image is loaded. Type **EFI_DEVICE_PATH** is defined in the **LocateDevicePath()** function description.
SourceBuffer	If not **NULL**, a pointer to the memory location containing a copy of the image to be loaded.
SourceSize	The size in bytes of *SourceBuffer*. Ignored if *SourceBuffer* is **NULL**.
ImageHandle	Pointer to the returned image handle that is created when the image is successfully loaded. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description.

Description

The `LoadImage()` function loads an EFI image into memory and returns a handle to the image. The image is loaded in one of two ways. If *SourceBuffer* is not **NULL**, the function is a memory-to-memory load in which *SourceBuffer* points to the image to be loaded and *SourceSize* indicates the image's size in bytes. In this case, the caller has copied the image into *SourceBuffer* and can free the buffer once loading is complete.

If *SourceBuffer* is **NULL**, the function is a file copy operation that uses the **SIMPLE_FILE_SYSTEM** protocol and then the **LOAD_FILE** protocol on the *DeviceHandle* to access the file referred to by *FilePath*. In this case, the *BootPolicy* flag is passed to the **LOAD_FILE.LoadFile()** function and is used to load the default image responsible for booting when the *FilePath* only indicates the device. For more information see the discussion of the Load File Protocol in Chapter 11.

Regardless of the type of load (memory-to-memory or file copy), the function relocates the code in the image while loading it.

Once the image is loaded, firmware creates and returns an **EFI_HANDLE** that identifies the image and supports the **EFI_LOADED_IMAGE** protocol. The caller may fill in the image's "load options" data, or add additional protocol support to the handle before passing control to the newly loaded image by calling **StartImage()**. Also, once the image is loaded, the caller either starts it by calling **StartImage()** or unloads it by calling **UnloadImage()**.

Status Codes Returned

EFI_SUCCESS	Image was loaded into memory correctly.
EFI_NOT_FOUND	The *FilePath* was not found.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_UNSUPPORTED	The image type is not supported, or the device path cannot be parsed to locate the proper protocol for loading the file.
EFI_OUT_OF_RESOURCES	Image was not loaded due to insufficient resources.
EFI_LOAD_ERROR	Image was not loaded because the image format was corrupt or not understood.
EFI_DEVICE_ERROR	Image was not loaded because the device returned a read error.

StartImage()

Summary

Transfers control to a loaded image's entry point.

Prototype

```
EFI_STATUS
StartImage (
      IN EFI_HANDLE          ImageHandle,
      OUT UINTN              *ExitDataSize,
      OUT CHAR16             **ExitData OPTIONAL
      );
```

Parameters

ImageHandle	Handle of image to be started. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description.
ExitDataSize	Pointer to the size, in bytes, of *ExitData*. If *ExitData* is **NULL**, then this parameter is ignored and the contents of *ExitDataSize* are not modified.
ExitData	Pointer to a pointer to a data buffer that includes a Null-terminated Unicode string, optionally followed by additional binary data. The string is a description that the caller may use to further indicate the reason for the image's exit.

Description

The **StartImage()** function transfers control to the entry point of an image that was loaded by **LoadImage()**. The image may only be started one time.

Control returns from **StartImage()** when the loaded image calls **Exit()**. When that call is made, the *ExitData* buffer and *ExitDataSize* from **Exit()** are passed back through the *ExitData* buffer and *ExitDataSize* in this function. The caller of this function is responsible for returning the *ExitData* buffer to the pool by calling **FreePool()** when the buffer is no longer needed.

EFI 1.10 Extension

To maintain compatibility with EFI drivers that are written to the *EFI 1.02 Specification*, **StartImage()** must monitor the handle database before and after each image is started. If any handles are created or modified when an image is started, then **ConnectController()** must be called for each of the newly created or modified handles before **StartImage()** returns.

Status Codes Returned

EFI_INVALID_PARAMETER	*ImageHandle* is not a handle to an unstarted image.
Exit code from image	Exit code from image.

UnloadImage()

Summary

Unloads an image.

Prototype

```
typedef
EFI_STATUS
UnloadImage (
  IN EFI_HANDLE   ImageHandle
  );
```

Parameters

ImageHandle	Handle that identifies the image to be unloaded.

Description

The **UnloadImage()** function unloads a previously loaded image.

There are three possible scenarios. If the image has not been started, the function unloads the image and returns **EFI_SUCCESS**.

If the image has been started and has an **Unload()** entry point, control is passed to that entry point. If the image's unload function returns **EFI_SUCCESS**, the image is unloaded; otherwise, the error returned by the image's unload function is returned to the caller. The image unload function is responsible for freeing all allocated memory and ensuring that there are no references to any freed memory, or to the image itself, before returning **EFI_SUCCESS**.

If the image has been started and does not have an **Unload()** entry point, the function returns **EFI_UNSUPPORTED**.

EFI 1.10 Extension

All of the protocols that were opened by *ImageHandle* using the boot service **OpenProtocol()** are automatically closed with the boot service **CloseProtocol()**. If all of the open protocols are closed, then **EFI_SUCCESS** is returned. If any call to **CloseProtocol()** fails, then the error code from **CloseProtocol()** is returned.

Status Codes Returned

EFI_SUCCESS	The image has been unloaded.
EFI_UNSUPPORTED	The image has been started, and does not support unload.
EFI_INVALID_PARAMETER	*ImageHandle* is not a valid image handle.
Exit code from Unload handler	Exit code from the image's unload function.

EFI_IMAGE_ENTRY_POINT

Summary

This is the declaration of an EFI image entry point. This can be the entry point to an EFI application, an EFI boot service driver, or an EFI runtime driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IMAGE_ENTRY_POINT) (
     IN EFI_HANDLE                 ImageHandle,
     IN EFI_SYSTEM_TABLE           *SystemTable
     );
```

Parameters

ImageHandle	Handle that identifies the loaded image. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description.
SystemTable	System Table for this image. Type **EFI_SYSTEM_TABLE** is defined in Chapter 4.

Description

An image's entry point is of type **EFI_IMAGE_ENTRY_POINT**. After firmware loads an image into memory, control is passed to the image's entry point. The entry point is responsible for initializing the image. The image's *ImageHandle* is passed to the image. The *ImageHandle* provides the image with all the binding and data information it needs. This information is available through protocol interfaces. However, to access the protocol interfaces on *ImageHandle* requires access to boot services functions. Therefore, **LoadImage()** passes to the **EFI_IMAGE_ENTRY_POINT** a *SystemTable* that is inherited from the current scope of **LoadImage()**.

All image handles support the **EFI_LOADED_IMAGE** protocol. This protocol can be used to obtain information about the loaded image's state—for example, the device from which the image was loaded and the image's load options. In addition, the *ImageHandle* may support other protocols provided by the parent image.

If the image supports dynamic unloading, it must supply an unload function in the **EFI_LOADED_IMAGE** structure before returning control from its entry point.

In general, an image returns control from its initialization entry point by calling **Exit()** or by returning control from its entry point. If the image returns control from its entry point, the firmware passes control to **Exit()** using the return code as the *ExitStatus* parameter to **Exit()**.

See **Exit()** below for entry point exit conditions.

Exit()

Summary

Terminates the currently loaded EFI image and returns control to boot services.

Prototype

```
typedef
EFI_STATUS
Exit (
  IN EFI_HANDLE   ImageHandle,
  IN EFI_STATUS   ExitStatus,
  IN UINTN        ExitDataSize,
  IN CHAR16       *ExitData      OPTIONAL
  );
```

Parameters

ImageHandle	Handle that identifies the image. This parameter is passed to the image on entry.
ExitStatus	The image's exit code.
ExitDataSize	The size, in bytes, of *ExitData*. Ignored if *ExitStatus* is **EFI_SUCCESS**.
ExitData	Pointer to a data buffer that includes a Null-terminated Unicode string, optionally followed by additional binary data. The string is a description that the caller may use to further indicate the reason for the image's exit. *ExitData* is only valid if *ExitStatus* is something other than **EFI_SUCCESS**. The *ExitData* buffer must be allocated by calling **AllocatePool()**.

Description

The **Exit()** function terminates the image referenced by *ImageHandle* and returns control to boot services. This function can only be called by the currently executing image. This function may not be called if the image has already returned from its entry point (**EFI_IMAGE_ENTRY_POINT**) or if it has loaded any child images that have not exited (all child images must exit before this image can exit).

Using **Exit()** is similar to returning from the image's **EFI_IMAGE_ENTRY_POINT** except that **Exit()** may also return additional *ExitData*.

When an EFI application exits, firmware frees the memory used to hold the image. The firmware also frees its references to the *ImageHandle* and the handle itself. Before exiting, the application is responsible for freeing any resources it allocated. This includes memory (pages and/or pool), open file system handles, and so forth. The only exception to this rule is the *ExitData* buffer, which must be freed by the caller of **StartImage()**. (If the buffer is needed, firmware must allocate it by calling **AllocatePool()** and must return a pointer to it to the caller of **StartImage()**.)

When an EFI boot service driver or runtime service driver exits, firmware frees the image only if the *ExitStatus* is an error code; otherwise the image stays resident in memory. The driver must not return an error code if it has installed any protocol handlers or other active callbacks into the system that have not (or cannot) be cleaned up. If the driver exits with an error code, it is responsible for freeing all resources before exiting. This includes any allocated memory (pages and/or pool), open file system handles, and so forth.

It is valid to call **Exit()** or **Unload()** for an image that was loaded by **LoadImage()** before calling **StartImage()**. This will free the image from memory without having started it.

EFI 1.10 Extension

If *ImageHandle* is an EFI Application, then all of the protocols that were opened by *ImageHandle* using the boot service **OpenProtocol()** are automatically closed with the boot service **CloseProtocol()**. If *ImageHandle* is an EFI boot services driver or runtime service driver, and *ExitStatus* is an error code, then all of the protocols that were opened by *ImageHandle* using the boot service **OpenProtocol()** are automatically closed with the boot service **CloseProtocol()**. If *ImageHandle* is an EFI boot services driver or runtime service driver, and *ExitStatus* is not an error code, then no protocols are automatically closed by this service.

Status Codes Returned

(Does not return.)	Image exit. Control is returned to the **StartImage()** call that invoked the image.
EFI_SUCCESS	The image was unloaded. **Exit()** only returns success if the image has not been started; otherwise, the exit returns to the **StartImage()** call that invoked the image.
EFI_INVALID_PARAMETER	The specified image is not the current image.

ExitBootServices()

Summary

Terminates all boot services.

Prototype

```
EFI_STATUS
ExitBootServices (
      IN EFI_HANDLE                    ImageHandle,
      IN UINTN                         MapKey
      );
```

Parameters

ImageHandle	Handle that identifies the exiting image. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description.
MapKey	Key to the latest memory map.

Description

The **ExitBootServices()** function is called by the currently executing EFI OS loader image to terminate all boot services. On success, the loader becomes responsible for the continued operation of the system.

An EFI OS loader must ensure that it has the system's current memory map at the time it calls **ExitBootServices()**. This is done by passing in the current memory map's *MapKey* value as returned by **GetMemoryMap()**. Care must be taken to ensure that the memory map does not change between these two calls. It is suggested that **GetMemoryMap()** be called immediately before calling **ExitBootServices()**.

On success, the EFI OS loader owns all available memory in the system. In addition, the loader can treat all memory in the map marked as **EfiBootServicesCode** and **EfiBootServicesData** as available free memory. No further calls to boot service functions or EFI device-handle-based protocols may be used, and the boot services watchdog timer is disabled. On success, several fields of the EFI System Table should be set to **NULL**. These include *ConsoleInHandle*, *ConIn*, *ConsoleOutHandle*, *ConOut*, *StandardErrorHandle*, *StdErr*, and *BootServicesTable*. In addition, since fields of the EFI System Table are being modified, the 32-bit CRC for the EFI System Table must be recomputed.

Status Codes Returned

EFI_SUCCESS	Boot services have been terminated.
EFI_INVALID_PARAMETER	*MapKey* is incorrect.

5.5 Miscellaneous Boot Services

This section contains the remaining function definitions for boot services not defined elsewhere but which are required to complete the definition of the EFI environment. Table 5-10 lists the Miscellaneous Boot Services Functions.

Table 5-10. Miscellaneous Boot Services Functions

Name	Type	Description
SetWatchDogTimer	Boot	Resets and sets a watchdog timer used during boot services time.
Stall	Boot	Stalls the processor.
CopyMem	Boot	Copies the contents of one buffer to another buffer.
SetMem	Boot	Fills a buffer with a specified value.
GetNextMonotonicCount	Boot	Returns a monotonically increasing count for the platform.
InstallConfigurationTable	Boot	Adds, updates, or removes a configuration table from the EFI System Table.
CalculateCrc32	Boot	Computes and returns a 32-bit CRC for a data buffer.

The **`CalculateCrc32()`** service was added because there are several places in EFI that 32-bit CRCs are used. These include the EFI System Table, the EFI Boot Services Table, the EFI Runtime Services Table, and the Guided Partition Table (GPT) structures. The **`CalculateCrc32()`** service allows new 32-bit CRCs to be computed, and existing 32-bit CRCs to be validated.

SetWatchdogTimer()

Summary

Sets the system's watchdog timer.

Prototype

```
EFI_STATUS
SetWatchdogTimer (
    IN UINTN        Timeout,
    IN UINT64       WatchdogCode,
    IN UINTN        DataSize,
    IN CHAR16       *WatchdogData    OPTIONAL
    );
```

Parameters

Timeout	The number of seconds to set the watchdog timer to. A value of zero disables the timer.
WatchdogCode	The numeric code to log on a watchdog timer timeout event. The firmware reserves codes 0x0000 to 0xFFFF. Loaders and operating systems may use other timeout codes.
DataSize	The size, in bytes, of *WatchdogData*.
WatchdogData	A data buffer that includes a Null-terminated Unicode string, optionally followed by additional binary data. The string is a description that the call may use to further indicate the reason to be logged with a watchdog event.

Description

The **SetWatchdogTimer()** function sets the system's watchdog timer.

If the watchdog timer expires, the event is logged by the firmware. The system may then either reset with the Runtime Service **ResetSystem()**, or perform a platform specific action that must eventually cause the platform to be reset. The watchdog timer is armed before the firmware's boot manager invokes an EFI boot option. The watchdog must be set to a period of 5 minutes. The EFI Image may reset or disable the watchdog timer as needed. If control is returned to the firmware's boot manager, the watchdog timer must be disabled.

The watchdog timer is only used during boot services. On successful completion of **ExitBootServices()** the watchdog timer is disabled.

The accuracy of the watchdog timer is +/- 1 second from the requested *Timeout*.

Status Codes Returned

EFI_SUCCESS	The timeout has been set.
EFI_INVALID_PARAMETER	The supplied *WatchdogCode* is invalid.
EFI_UNSUPPORTED	The system does not have a watchdog timer.
EFI_DEVICE_ERROR	The watch dog timer could not be programmed due to a hardware error.

Stall()

Summary

Induces a fine-grained stall.

Prototype

```
EFI_STATUS
Stall (
      IN UINTN                      Microseconds
      )
```

Parameters

Microseconds	The number of microseconds to stall execution.

Description

The **Stall()** function stalls execution on the processor for at least the requested number of microseconds. Execution of the processor is *not* yielded for the duration of the stall.

Status Codes Returned

EFI_SUCCESS	Execution was stalled at least the requested number of *Microseconds*.

CopyMem()

Summary

The **CopyMem()** function copies the contents of one buffer to another buffer.

Prototype

```
VOID
CopyMem (
  IN VOID      *Destination,
  IN VOID      *Source,
  IN UINTN     Length
  );
```

Parameters

Destination	Pointer to the destination buffer of the memory copy.
Source	Pointer to the source buffer of the memory copy.
Length	Number of bytes to copy from *Source* to *Destination*.

Description

The **CopyMem()** function copies *Length* bytes from the buffer *Source* to the buffer *Destination*.

The implementation of **CopyMem()** must be reentrant, and it must handle overlapping *Source* and *Destination* buffers. This means that the implementation of **CopyMem()** must choose the correct direction of the copy operation based on the type of overlap that exists between the *Source* and *Destination* buffers. If either the *Source* buffer or the *Destination* buffer crosses the top of the processor's address space, then the result of the copy operation is unpredictable.

The contents of the *Destination* buffer on exit from this service must match the contents of the *Source* buffer on entry to this service. Due to potential overlaps, the contents of the *Source* buffer may be modified by this service. The following rules can be used to guarantee the correct behavior:

1. If *Destination* and *Source* are identical, then no operation should be performed.
2. If *Destination* > *Source* **and** *Destination* < (*Source* + *Length*), then the data should be copied from the *Source* buffer to the *Destination* buffer starting from the end of the buffers and working toward the beginning of the buffers.
3. Otherwise, the data should be copied from the *Source* buffer to the *Destination* buffer starting from the beginning of the buffers and working toward the end of the buffers.

Status Codes Returned

None.

SetMem()

Summary

The **SetMem()** function fills a buffer with a specified value.

Prototype

```
VOID
SetMem (
  IN VOID      *Buffer,
  IN UINTN     Size,
  IN UINT8     Value
  );
```

Parameters

Buffer	Pointer to the buffer to fill.
Size	Number of bytes in *Buffer* to fill.
Value	Value to fill *Buffer* with.

Description

This function fills *Size* bytes of *Buffer* with *Value*. The implementation of **SetMem()** must be reentrant. If *Buffer* crosses the top of the processor's address space, the result of the **SetMem()** operation is unpredictable.

Status Codes Returned

None.

GetNextMonotonicCount()

Summary

Returns a monotonically increasing count for the platform.

Prototype

```
EFI_STATUS
GetNextMonotonicCount (
      OUT UINT64             *Count
      );
```

Parameters

Count	Pointer to returned value.

Description

The **GetNextMonotonicCount()** function returns a 64-bit value that is numerically larger then the last time the function was called.

The platform's monotonic counter is comprised of two parts: the high 32 bits and the low 32 bits. The low 32-bit value is volatile and is reset to zero on every system reset. It is increased by 1 on every call to **GetNextMonotonicCount()**. The high 32-bit value is nonvolatile and is increased by one on whenever the system resets or the low 32-bit counter overflows.

Status Codes Returned

EFI_SUCCESS	The next monotonic count was returned.
EFI_DEVICE_ERROR	The device is not functioning properly.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

InstallConfigurationTable()

Summary

Adds, updates, or removes a configuration table entry from the EFI System Table.

Prototype

```
EFI_STATUS
InstallConfigurationTable (
      IN EFI_GUID                *Guid,
      IN VOID                    *Table
      );
```

Parameters

Guid	A pointer to the GUID for the entry to add, update, or remove.
Table	A pointer to the configuration table for the entry to add, update, or remove. May be **NULL**.

Description

The **InstallConfigurationTable()** function is used to maintain the list of configuration tables that are stored in the EFI System Table. The list is stored as an array of (GUID, Pointer) pairs. The list must be allocated from pool memory with *PoolType* set to **EfiRuntimeServicesData**.

If *Guid* is not a valid GUID, **EFI_INVALID_PARAMETER** is returned. If *Guid* is valid, there are four possibilities:

- If *Guid* is not present in the System Table, and *Table* is not **NULL**, then the (*Guid*, *Table*) pair is added to the System Table. See Note below.
- If *Guid* is not present in the System Table, and *Table* is **NULL**, then **EFI_NOT_FOUND** is returned.
- If *Guid* is present in the System Table, and *Table* is not **NULL**, then the (*Guid*, *Table*) pair is updated with the new *Table* value.
- If *Guid* is present in the System Table, and *Table* is **NULL**, then the entry associated with *Guid* is removed from the System Table.

If an add, modify, or remove operation is completed, then **EFI_SUCCESS** is returned.

NOTE

If there is not enough memory to perform an add operation, then **EFI_OUT_OF_RESOURCES** *is returned.*

Status Codes Returned

EFI_SUCCESS	The (*Guid*, *Table*) pair was added, updated, or removed.
EFI_INVALID_PARAMETER	*Guid* is not valid.
EFI_NOT_FOUND	An attempt was made to delete a nonexistent entry.
EFI_OUT_OF_RESOURCES	There is not enough memory available to complete the operation.

CalculateCrc32()

Summary

Computes and returns a 32-bit CRC for a data buffer.

Prototype

```
typedef
EFI_STATUS
CalculateCrc32 (
  IN  VOID     *Data,
  IN  UINTN    DataSize,
  OUT UINT32   *Crc32
  );
```

Parameters

Data	A pointer to the buffer on which the 32-bit CRC is to be computed.
DataSize	The number of bytes in the buffer *Data*.
Crc32	The 32-bit CRC that was computed for the data buffer specified by *Data* and *DataSize*.

Description

This function computes the 32-bit CRC for the data buffer specified by *Data* and *DataSize*. If the 32-bit CRC is computed, then it is returned in *Crc32* and **EFI_SUCCESS** is returned.

If *Data* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *Crc32* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *DataSize* is 0, then **EFI_INVALID_PARAMETER** is returned.

Status Codes Returned

EFI_SUCCESS	The 32-bit CRC was computed for the data buffer and returned in *Crc32*.
EFI_INVALID_PARAMETER	*Data* is **NULL**.
EFI_INVALID_PARAMETER	*Crc32* is **NULL**.
EFI_INVALID_PARAMETER	*DataSize* is 0.

6
Services - Runtime Services

This chapter discusses the fundamental services that are present in an EFI-compliant system. The services are defined by interface functions that may be used by code running in the EFI environment. Such code may include protocols that manage device access or extend platform capability, as well as EFI applications running in the preboot environment and EFI OS loaders.

Two types of services are described here:

- **Boot Services**. Functions that are available *before* a successful call to `ExitBootServices()`. These functions are described in Chapter 5.
- **Runtime Services**. Functions that are available *before and after* any call to `ExitBootServices()`. These functions are described in this chapter.

During boot, system resources are owned by the firmware and are controlled through boot services interface functions. These functions can be characterized as “global” or “handle-based.” The term “global” simply means that a function accesses system services and is available on all platforms (since all platforms support all system services). The term “handle-based” means that the function accesses a specific device or device functionality and may not be available on some platforms (since some devices are not available on some platforms). Protocols are created dynamically. This chapter discusses the “global” functions and runtime functions; subsequent chapters discuss the “handle-based.”

EFI applications (including OS loaders) must use boot services functions to access devices and allocate memory. On entry, an EFI Image is provided a pointer to an EFI system table which contains the Boot Services dispatch table and the default handles for accessing the console. All boot services functionality is available until an EFI OS loader loads enough of its own environment to take control of the system’s continued operation and then terminates boot services with a call to `ExitBootServices()`.

In principle, the `ExitBootServices()` call is intended for use by the operating system to indicate that its loader is ready to assume control of the platform and all platform resource management. Thus boot services are available up to this point to assist the OS loader in preparing to boot the operating system. Once the OS loader takes control of the system and completes the operating system boot process, only runtime services may be called. Code other than the OS loader, however, may or may not choose to call `ExitBootServices()`. This choice may in part depend upon whether or not such code is designed to make continued use of EFI boot services or the boot services environment.

The rest of this chapter discusses individual functions. Runtime Services fall into these categories:

- Variable Services (Section 6.1)
- Time Services (Section 6.2)
- Virtual Memory Services (Section 6.3)
- Miscellaneous Services (Section 6.4)

6.1 Variable Services

Variables are defined as key/value pairs that consist of identifying information plus attributes (the key) and arbitrary data (the value). Variables are intended for use as a means to store data that is passed between the EFI environment implemented in the platform and EFI OS loaders and other applications that run in the EFI environment.

Although the implementation of variable storage is not defined in this specification, variables must be persistent in most cases. This implies that the EFI implementation on a platform must arrange it so that variables passed in for storage are retained and available for use each time the system boots, at least until they are explicitly deleted or overwritten. Provision of this type of nonvolatile storage may be very limited on some platforms, so variables should be used sparingly in cases where other means of communicating information cannot be used.

Table 6-1 lists the variable services functions described in this section:

Table 6-1. Variable Services Functions

Name	Type	Description
GetVariable	Runtime	Returns the value of a variable.
GetNextVariableName	Runtime	Enumerates the current variable names.
SetVariable	Runtime	Sets the value of a variable.

GetVariable()

Summary

Returns the value of a variable.

Prototype

```
EFI_STATUS
GetVariable (
     IN CHAR16              *VariableName,
     IN EFI_GUID            *VendorGuid,
     OUT UINT32             *Attributes OPTIONAL,
     IN OUT UINTN           *DataSize,
     OUT VOID               *Data
     );
```

Parameters

VariableName	A Null-terminated Unicode string that is the name of the vendor's variable.
VendorGuid	A unique identifier for the vendor. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.
Attributes	If not **NULL**, a pointer to the memory location to return the attributes bitmask for the variable. See "Related Definitions."
DataSize	On input, the size in bytes of the return *Data* buffer. On output the size of data returned in *Data*.
Data	The buffer to return the contents of the variable.

Related Definitions

```
//*******************************************************
// Variable Attributes
//*******************************************************
#define EFI_VARIABLE_NON_VOLATILE          0x0000000000000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS    0x0000000000000002
#define EFI_VARIABLE_RUNTIME_ACCESS        0x0000000000000004
```

Description

Each vendor may create and manage its own variables without the risk of name conflicts by using a unique *VendorGuid*. When a variable is set its *Attributes* are supplied to indicate how the data variable should be stored and maintained by the system. The attributes affect when the variable may be accessed and volatility of the data. Any attempts to access a variable that does not have the attribute set for runtime access will yield the **EFI_NOT_FOUND** error.

If the *Data* buffer is too small to hold the contents of the variable, the error **EFI_BUFFER_TOO_SMALL** is returned and *DataSize* is set to the required buffer size to obtain the data.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The variable was not found.
EFI_BUFFER_TOO_SMALL	The *DataSize* is too small for the result. *DataSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	*VariableName* is **NULL**.
EFI_INVALID_PARAMETER	*VendorGuid* is **NULL**.
EFI_INVALID_PARAMETER	*DataSize* is **NULL**.
EFI_INVALID_PARAMETER	*Data* is **NULL**.
EFI_DEVICE_ERROR	The variable could not be retrieved due to a hardware error.

GetNextVariableName()

Summary

Enumerates the current variable names.

Prototype

```
EFI_STATUS
GetNextVariableName (
      IN OUT UINTN          *VariableNameSize,
      IN OUT CHAR16         *VariableName,
      IN OUT EFI_GUID       *VendorGuid
      );
```

Parameters

VariableNameSize	The size of the *VariableName* buffer.
VariableName	On input, supplies the last *VariableName* that was returned by **GetNextVariableName()**. On output, returns the Null-terminated Unicode string of the current variable.
VendorGuid	On input, supplies the last *VendorGuid* that was returned by **GetNextVariableName()**. On output, returns the *VendorGuid* of the current variable. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.

Description

GetNextVariableName() is called multiple times to retrieve the *VariableName* and *VendorGuid* of all variables currently available in the system. On each call to **GetNextVariableName()** the previous results are passed into the interface, and on output the interface returns the next variable name data. When the entire variable list has been returned, the error **EFI_NOT_FOUND** is returned.

Note that if **EFI_BUFFER_TOO_SMALL** is returned, the *VariableName* buffer was too small for the next variable. When such an error occurs, the *VariableNameSize* is updated to reflect the size of buffer needed. In all cases when calling **GetNextVariableName()** the *VariableNameSize* must not exceed the actual buffer size that was allocated for *VariableName*.

To start the search, a Null-terminated string is passed in *VariableName*; that is, *VariableName* is a pointer to a Null Unicode character. This is always done on the initial call to **GetNextVariableName()**. When *VariableName* is a pointer to a Null Unicode character, *VendorGuid* is ignored. **GetNextVariableName()** cannot be used as a filter to return variable names with a specific GUID. Instead, the entire list of variables must be retrieved, and the caller may act as a filter if it chooses. Calls to **SetVariable()** between calls to **GetNextVariableName()** may produce unpredictable results.

Once **ExitBootServices()** is performed, variables that are only visible during boot services will no longer be returned. To obtain the data contents or attribute for a variable returned by **GetNextVariableName()**, the **GetVariable()** interface is used.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The next variable was not found.
EFI_BUFFER_TOO_SMALL	The *VariableNameSize* is too small for the result. *VariableNameSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	*VariableNameSize* is **NULL**.
EFI_INVALID_PARAMETER	*VariableName* is **NULL**.
EFI_INVALID_PARAMETER	*VendorGuid* is **NULL**.
EFI_DEVICE_ERROR	The variable name could not be retrieved due to a hardware error.

SetVariable()

Summary

Sets the value of a variable.

Prototype

```
EFI_STATUS
SetVariable (
      IN CHAR16               *VariableName,
      IN EFI_GUID             *VendorGuid,
      IN UINT32               Attributes,
      IN UINTN                DataSize,
      IN VOID                 *Data
      );
```

Parameters

VariableName	A Null-terminated Unicode string that is the name of the vendor's variable. Each *VariableName* is unique for each *VendorGuid*. *VariableName* must contain 1 or more Unicode characters. If *VariableName* is an empty Unicode string, then **EFI_INVALID_PARAMETER** is returned.
VendorGuid	A unique identifier for the vendor. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.
Attributes	Attributes bitmask to set for the variable. Refer to the **GetVariable()** function description.
DataSize	The size in bytes of the *Data* buffer. A size of zero causes the variable to be deleted.
Data	The contents for the variable.

Description

Variables are stored by the firmware and may maintain their values across power cycles. Each vendor may create and manage its own variables without the risk of name conflicts by using a unique *VendorGuid*.

Each variable has *Attributes* that define how the firmware stores and maintains the data value. If the **EFI_VARIABLE_NON_VOLATILE** attribute is *not* set, the firmware stores the variable in normal memory and it is not maintained across a power cycle. Such variables are used to pass information from one component to another. An example of this is the firmware's language code support variable. It is created at firmware initialization time for access by EFI components that may need the information, but does not need to be backed up to nonvolatile storage.

`EFI_VARIABLE_NON_VOLATILE` variables are stored in fixed hardware that has a limited storage capacity; sometimes a severely limited capacity. Software should only use a nonvolatile variable when absolutely necessary. In addition, if software uses a nonvolatile variable it should use a variable that is only accessible at boot services time if possible.

A variable must contain one or more bytes of *`Data`*. Using **`SetVariable()`** with a *`DataSize`* of zero causes the entire variable to be deleted. The space consumed by the deleted variable may not be available until the next power cycle.

The Attributes have the following usage rules:

- Storage attributes are only applied to a variable when creating the variable. If a preexisting variable is rewritten with different attributes, the result is indeterminate and may vary between implementations. The correct method of changing the attributes of a variable is to delete the variable and recreate it with different attributes. There is one exception to this rule. If a preexisting variable is rewritten with no access attributes specified, the variable will be deleted.
- Setting a data variable with no access, or zero *`DataSize`* attributes specified causes it to be deleted.
- Runtime access to a data variable implies boot service access. Attributes that have **`EFI_VARIABLE_RUNTIME_ACCESS`** set must also have **`EFI_VARIABLE_BOOTSERVICE_ACCESS`** set. The caller is responsible for following this rule.
- Once **`ExitBootServices()`** is performed, data variables that did not have **`EFI_VARIABLE_RUNTIME_ACCESS`** set are no longer visible to **`GetVariable()`**.
- Once **`ExitBootServices()`** is performed, only variables that have **`EFI_VARIABLE_RUNTIME_ACCESS`** and **`EFI_VARIABLE_NON_VOLATILE`** set can be set with **`SetVariable()`**. Variables that have runtime access but that are not nonvolatile are read-only data variables once **`ExitBootServices()`** is performed.

The only rules the firmware must implement when saving a nonvolatile variable is that it has actually been saved to nonvolatile storage before returning **`EFI_SUCCESS,`** and that a partial save is not performed. If power fails during a call to **`SetVariable()`** the variable may contain its previous value, or its new value. In addition there is no read, write, or delete security protection.

Status Codes Returned

EFI_SUCCESS	The firmware has successfully stored the variable and its data as defined by the Attributes.
EFI_INVALID_PARAMETER	An invalid combination of attribute bits was supplied, or the *`DataSize`* exceeds the maximum allowed.
EFI_INVALID_PARAMETER	*`VariableName`* is an empty Unicode string.
EFI_OUT_OF_RESOURCES	Not enough storage is available to hold the variable and its data.
EFI_DEVICE_ERROR	The variable could not be saved due to a hardware failure.

6.2 Time Services

This section contains function definitions for time-related functions that are typically needed by operating systems at runtime to access underlying hardware that manages time information and services. The purpose of these interfaces is to provide operating system writers with an abstraction for hardware time devices, thereby relieving the need to access legacy hardware devices directly. There is also a stalling function for use in the preboot environment. Table 6-2 lists the time services functions described in this section:

Table 6-2. Time Services Functions

Name	Type	Description
GetTime	Runtime	Returns the current time and date, and the time-keeping capabilities of the platform.
SetTime	Runtime	Sets the current local time and date information.
GetWakeupTime	Runtime	Returns the current wakeup alarm clock setting.
SetWakeupTime	Runtime	Sets the system wakeup alarm clock time.

GetTime()

Summary

Returns the current time and date information, and the time-keeping capabilities of the hardware platform.

Prototype

```
EFI_STATUS
GetTime (
     OUT EFI_TIME                    *Time,
     OUT EFI_TIME_CAPABILITIES       *Capabilities OPTIONAL
     );
```

Parameters

Time	A pointer to storage to receive a snapshot of the current time. Type **EFI_TIME** is defined in "Related Definitions."
Capabilities	An optional pointer to a buffer to receive the real time clock device's capabilities. Type **EFI_TIME_CAPABILITIES** is defined in "Related Definitions."

Related Definitions

```
//*******************************************************
//EFI_TIME
//*******************************************************
// This represents the current time information
typedef struct {
     UINT16          Year;            // 1998 – 20XX
     UINT8           Month;           // 1 – 12
     UINT8           Day;             // 1 – 31
     UINT8           Hour;            // 0 – 23
     UINT8           Minute;          // 0 – 59
     UINT8           Second;          // 0 – 59
     UINT8           Pad1;
     UINT32          Nanosecond;      // 0 – 999,999,999
     INT16           TimeZone;        // -1440 to 1440 or 2047
     UINT8           Daylight;
     UINT8           Pad2;
} EFI_TIME;
```

```
//*******************************************************
// Bit Definitions for EFI_TIME.Daylight.  See below.
//*******************************************************
#define EFI_TIME_ADJUST_DAYLIGHT      0x01
#define EFI_TIME_IN_DAYLIGHT          0x02

//*******************************************************
// Value Definition for EFI_TIME.TimeZone.  See below.
//*******************************************************
#define EFI_UNSPECIFIED_TIMEZONE      0x07FF
```

Year, Month, Day — The current local date.

Hour, Minute, Second, Nanosecond

The current local time. Nanoseconds report the current fraction of a second in the device. The format of the time is *hh:mm:ss.nnnnnnnnn*. A battery backed real time clock device maintains the date and time.

TimeZone — The time's offset in minutes from GMT. If the value is **EFI_UNSPECIFIED_TIMEZONE**, then the time is interpreted as a local time.

Daylight — A bitmask containing the daylight savings time information for the time.

The **EFI_TIME_ADJUST_DAYLIGHT** bit indicates if the time is affected by daylight savings time or not. This value does not indicate that the time has been adjusted for daylight savings time. It indicates only that it should be adjusted when the **EFI_TIME** enters daylight savings time.

If **EFI_TIME_IN_DAYLIGHT** is set, the time has been adjusted for daylight savings time.

All other bits must be zero.

```
//*******************************************************
// EFI_TIME_CAPABILITIES
//*******************************************************
// This provides the capabilities of the
// real time clock device as exposed through the EFI interfaces.
typedef struct {
     UINT32           Resolution;
     UINT32           Accuracy;
     BOOLEAN          SetsToZero;
} EFI_TIME_CAPABILITIES;
```

Resolution	Provides the reporting resolution of the real-time clock device in counts per second. For a normal PC-AT CMOS RTC device, this value would be 1 Hz, or 1, to indicate that the device only reports the time to the resolution of 1 second.
Accuracy	Provides the timekeeping accuracy of the real-time clock in an error rate of 1E-6 parts per million. For a clock with an accuracy of 50 parts per million, the value in this field would be 50,000,000.
SetsToZero	A **TRUE** indicates that a time set operation clears the device's time below the *Resolution* reporting level. A **FALSE** indicates that the state below the *Resolution* level of the device is not cleared when the time is set. Normal PC-AT CMOS RTC devices set this value to **FALSE**.

Description

The **GetTime()** function returns a time that was valid sometime during the call to the function. While the returned **EFI_TIME** structure contains *TimeZone* and *Daylight* savings time information, the actual clock does not maintain these values. The current time zone and daylight saving time information returned by **GetTime()** are the values that were last set via **SetTime()**.

The **GetTime()** function should take approximately the same amount of time to read the time each time it is called. All reported device capabilities are to be rounded up.

During runtime, if a PC-AT CMOS device is present in the platform the caller must synchronize access to the device before calling **GetTime()**.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully.
EFI_INVALID_PARAMETER	*Time* is **NULL**.
EFI_DEVICE_ERROR	The time could not be retrieved due to a hardware error.

SetTime()

Summary

Sets the current local time and date information.

Prototype

```
EFI_STATUS
SetTime (
      IN EFI_TIME            *Time
      );
```

Parameters

Time	A pointer to the current time. Type **EFI_TIME** is defined in the **GetTime()** function description. Full error checking is performed on the different fields of the **EFI_TIME** structure (refer to the **EFI_TIME** definition in the **GetTime()** function description for full details), and **EFI_INVALID_PARAMETER** is returned if any field is out of range.

Description

The **SetTime()** function sets the real time clock device to the supplied time, and records the current time zone and daylight savings time information. The **SetTime()** function is not allowed to loop based on the current time. For example, if the device does not support a hardware reset for the sub-resolution time, the code is *not* to implement the feature by waiting for the time to wrap.

During runtime, if a PC-AT CMOS device is present in the platform the caller must synchronize access to the device before calling **SetTime()**.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully.
EFI_INVALID_PARAMETER	A time field is out of range.
EFI_DEVICE_ERROR	The time could not be set due to a hardware error.

GetWakeupTime()

Summary

Returns the current wakeup alarm clock setting.

Prototype

```
EFI_STATUS
GetWakeupTime (
      OUT BOOLEAN      *Enabled,
      OUT BOOLEAN      *Pending,
      OUT EFI_TIME     *Time
      );
```

Parameters

Enabled	Indicates if the alarm is currently enabled or disabled.
Pending	Indicates if the alarm signal is pending and requires acknowledgement.
Time	The current alarm setting. Type **EFI_TIME** is defined in the **GetTime()** function description.

Description

The alarm clock time may be rounded from the set alarm clock time to be within the resolution of the alarm clock device. The resolution of the alarm clock device is defined to be one second.

During runtime, if a PC-AT CMOS device is present in the platform the caller must synchronize access to the device before calling **GetWakeupTime()**.

Status Codes Returned

EFI_SUCCESS	The alarm settings were returned.
EFI_INVALID_PARAMETER	*Enabled* is **NULL**.
EFI_INVALID_PARAMETER	*Pending* is **NULL**.
EFI_INVALID_PARAMETER	*Time* is **NULL**.
EFI_DEVICE_ERROR	The wakeup time could not be retrieved due to a hardware error.
EFI_UNSUPPORTED	A wakeup timer is not supported on this platform.

SetWakeupTime()

Summary

Sets the system wakeup alarm clock time.

Prototype

```
EFI_STATUS
SetWakeupTime (
    IN BOOLEAN      Enable,
    IN EFI_TIME     *Time          OPTIONAL
    );
```

Parameters

Enable	Enable or disable the wakeup alarm.
Time	If *Enable* is **TRUE**, the time to set the wakeup alarm for. Type **EFI_TIME** is defined in the **GetTime()** function description. If *Enable* is **FALSE**, then this parameter is optional, and may be **NULL**.

Description

Setting a system wakeup alarm causes the system to wake up or power on at the set time. When the alarm fires, the alarm signal is latched until acknowledged by calling **SetWakeupTime()** to disable the alarm. If the alarm fires before the system is put into a sleeping or off state, since the alarm signal is latched the system will immediately wake up. If the alarm fires while the system is off and there is insufficient power to power on the system, the system is powered on when power is restored.

For an ACPI-aware operating system, this function only handles programming the wakeup alarm for the desired wakeup time. The operating system still controls the wakeup event as it normally would through the ACPI Power Management register set.

The resolution for the wakeup alarm is defined to be 1 second.

During runtime, if a PC-AT CMOS device is present in the platform the caller must synchronize access to the device before calling **SetWakeupTime()**.

Status Codes Returned

EFI_SUCCESS	If *Enable* is **TRUE**, then the wakeup alarm was enabled. If *Enable* is **FALSE**, then the wakeup alarm was disabled.
EFI_INVALID_PARAMETER	A time field is out of range.
EFI_DEVICE_ERROR	The wakeup time could not be set due to a hardware error.
EFI_UNSUPPORTED	A wakeup timer is not supported on this platform.

6.3 Virtual Memory Services

This section contains function definitions for the virtual memory support that may be optionally used by an operating system at runtime. If an operating system chooses to make EFI runtime service calls in a virtual addressing mode instead of the flat physical mode, then the operating system must use the services in this section to switch the EFI runtime services from flat physical addressing to virtual addressing. Table 6-3 lists the virtual memory service functions described in this section. The system firmware must follow the processor-specific rules outlined in sections 2.3.2 and 2.3.3 in the layout of the EFI memory map to enable the OS to make the required virtual mappings.

Table 6-3. Virtual Memory Functions

Name	Type	Description
SetVirtualAddressMap	Runtime	Used by an OS loader to convert from physical addressing to virtual addressing.
ConvertPointer	Runtime	Used by EFI components to convert internal pointers when switching to virtual addressing.

SetVirtualAddressMap()

Summary

Changes the runtime addressing mode of EFI firmware from physical to virtual.

Prototype

```
EFI_STATUS
SetVirtualAddressMap (
      IN UINTN                       MemoryMapSize,
      IN UINTN                       DescriptorSize,
      IN UINT32                      DescriptorVersion,
      IN EFI_MEMORY_DESCRIPTOR       *VirtualMap
      );
```

Parameters

MemoryMapSize	The size in bytes of *VirtualMap*.
DescriptorSize	The size in bytes of an entry in the *VirtualMap*.
DescriptorVersion	The version of the structure entries in *VirtualMap*.
VirtualMap	An array of memory descriptors which contain new virtual address mapping information for all runtime ranges. Type **EFI_MEMORY_DESCRIPTOR** is defined in the **GetMemoryMap()** function description.

Description

The **SetVirtualAddressMap()** function is used by the OS loader. The function can only be called at runtime, and is called by the owner of the system's memory map. I.e., the component which called **ExitBootServices()**.

This call changes the addresses of the runtime components of the EFI firmware to the new virtual addresses supplied in the *VirtualMap*. The supplied *VirtualMap* must provide a new virtual address for every entry in the memory map at **ExitBootServices()** that is marked as being needed for runtime usage. All of the virtual address fields in the *VirtualMap* must be aligned on 4 KB boundaries.

The call to **SetVirtualAddressMap()** must be done with the physical mappings. On successful return from this function, the system must then make any future calls with the newly assigned virtual mappings. All address space mappings must be done in accordance to the cacheability flags as specified in the original address map.

When this function is called, all events that were registered to be signaled on an address map change are notified. Each component that is notified must update any internal pointers for their new addresses. This can be done with the **`ConvertPointer()`** function. Once all events have been notified, the EFI firmware reapplies image "fix-up" information to virtually relocate all runtime images to their new addresses. In addition, all of the fields of the EFI Runtime Services Table except *`SetVirtualAddressMap`* and *`ConvertPointer`* must be converted from physical pointers to virtual pointers using the **`ConvertPointer()`** service. The **`SetVirtualAddressMap()`** and **`ConvertPointer()`** services are only callable in physical mode, so they do not need to be converted from physical pointers to virtual pointers. Several fields of the EFI System Table must be converted from physical pointers to virtual pointers using the **`ConvertPointer()`** service. These fields include *`FirmwareVendor`*, *`RuntimeServices`*, and *`ConfigurationTable`*. Because contents of both the EFI Runtime Services Table and the EFI System Table are modified by this service, the 32-bit CRC for the EFI Runtime Services Table and the EFI System Table must be recomputed.

A virtual address map may only be applied one time. Once the runtime system is in virtual mode, calls to this function return **`EFI_UNSUPPORTED`**.

Status Codes Returned

EFI_SUCCESS	The virtual address map has been applied.
EFI_UNSUPPORTED	EFI firmware is not at runtime, or the EFI firmware is already in virtual address mapped mode.
EFI_INVALID_PARAMETER	*`DescriptorSize`* or *`DescriptorVersion`* is invalid.
EFI_NO_MAPPING	A virtual address was not supplied for a range in the memory map that requires a mapping.
EFI_NOT_FOUND	A virtual address was supplied for an address that is not found in the memory map.

ConvertPointer()

Summary

Determines the new virtual address that is to be used on subsequent memory accesses.

Prototype

```
EFI_STATUS
ConvertPointer (
      IN UINTN          DebugDisposition,
      IN VOID           **Address
      );
```

Parameters

DebugDisposition	Supplies type information for the pointer being converted. See "Related Definitions."
Address	A pointer to a pointer that is to be fixed to be the value needed for the new virtual address mappings being applied.

Related Definitions

```
//*******************************************************
// EFI_OPTIONAL_PTR
//*******************************************************
#define EFI_OPTIONAL_PTR            0x00000001
```

Description

The **ConvertPointer()** function is used by an EFI component during the **SetVirtualAddressMap()** operation.

The **ConvertPointer()** function updates the current pointer pointed to by *Address* to be the proper value for the new address map. Only runtime components need to perform this operation. The **CreateEvent()** function is used to create an event that is to be notified when the address map is changing. All pointers the component has allocated or assigned must be updated.

If the **EFI_OPTIONAL_PTR** flag is specified, the pointer being converted is allowed to be **NULL**.

Once all components have been notified of the address map change, firmware fixes any compiled in pointers that are embedded in any runtime image.

Status Codes Returned

EFI_SUCCESS	The pointer pointed to by *Address* was modified.
EFI_NOT_FOUND	The pointer pointed to by *Address* was not found to be part of the current memory map. This is normally fatal.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

6.4 Miscellaneous Runtime Services

This section contains the remaining function definitions for runtime services not defined elsewhere but which are required to complete the definition of the EFI environment. Table 6-4 lists the Miscellaneous Runtime Services.

Table 6-4. Miscellaneous Runtime Services

Name	Type	Description
ResetSystem	Runtime	Resets the entire platform.
GetNextHighMonotonicCount	Runtime	Returns the next high 32 bits of the platform's monotonic counter.

ResetSystem()

Summary

Resets the entire platform.

Prototype

```
VOID
ResetSystem (
      IN EFI_RESET_TYPE       ResetType,
      IN EFI_STATUS           ResetStatus,
      IN UINTN                DataSize,
      IN CHAR16               *ResetData OPTIONAL
      );
```

Parameters

ResetType	The type of reset to perform. Type **EFI_RESET_TYPE** is defined in “Related Definitions” below.
ResetStatus	The status code for the reset. If the system reset is part of a normal operation, the status code would be **EFI_SUCCESS**. If the system reset is due to some type of failure the most appropriate EFI Status code would be used.
DataSize	The size, in bytes, of *ResetData*.
ResetData	A data buffer that includes a Null-terminated Unicode string, optionally followed by additional binary data. The string is a description that the caller may use to further indicate the reason for the system reset. *ResetData* is only valid if *ResetStatus* is something other then **EFI_SUCCESS**. This pointer must be a physical address.

Related Definitions

```
//*******************************************************
// EFI_RESET_TYPE
//*******************************************************
typedef enum {
      EfiResetCold,
      EfiResetWarm,
      EfiResetShutdown
} EFI_RESET_TYPE;
```

Description

The **`ResetSystem()`** function resets the entire platform, including all processors and devices, and reboots the system.

Calling this interface with *`ResetType`* of **`EfiResetCold`** causes a system-wide reset. This sets all circuitry within the system to its initial state. This type of reset is asynchronous to system operation and operates without regard to cycle boundaries. **`EfiResetCold`** is tantamount to a system power cycle.

Calling this interface with *`ResetType`* of **`EfiResetWarm`** causes a system-wide initialization. The processors are set to their initial state, and pending cycles are not corrupted. If the system does not support this reset type, then an **`EfiResetCold`** must be performed.

Calling this interface with *`ResetType`* of **`EfiResetShutdown`** causes the system to enter a power state equivalent to the ACPI G2/S5 or G3 states. If the system does not support this reset type, then when the system is rebooted, it should exhibit the **`EfiResetCold`** attributes. If the ACPI S5 state is supported on the system, then this reset type should not be used.

The platform may optionally log the parameters from any non-normal reset that occurs.

The **`ResetSystem()`** function does not return.

GetNextHighMonotonicCount()

Summary

Returns the next high 32 bits of the platform's monotonic counter.

Prototype

```
EFI_STATUS
GetNextHighMonotonicCount (
      OUT UINT32            *HighCount
      );
```

Parameters

HighCount	Pointer to returned value.

Description

The **GetNextHighMonotonicCount()** function returns the next high 32 bits of the platform's monotonic counter.

The platform's monotonic counter is comprised of two 32-bit quantities: the high 32 bits and the low 32 bits. During boot service time the low 32-bit value is volatile: it is reset to zero on every system reset and is increased by 1 on every call to **GetNextMonotonicCount()**. The high 32-bit value is nonvolatile and is increased by 1 whenever the system resets or whenever the low 32-bit count (returned by **GetNextMonoticCount()**) overflows.

The **GetNextMonotonicCount()** function is only available at boot services time. If the operating system wishes to extend the platform monotonic counter to runtime, it may do so by utilizing **GetNextHighMonotonicCount()**. To do this, before calling **ExitBootServices()** the operating system would call **GetNextMonotonicCount()** to obtain the current platform monotonic count. The operating system would then provide an interface that returns the next count by:

- Adding 1 to the last count.
- Before the lower 32 bits of the count overflows, call **GetNextHighMonotonicCount()**. This will increase the high 32 bits of the platform's nonvolatile portion of the monotonic count by 1.

This function may only be called at Runtime.

Status Codes Returned

EFI_SUCCESS	The next high monotonic count was returned.
EFI_DEVICE_ERROR	The device is not functioning properly.
EFI_INVALID_PARAMETER	*HighCount* is **NULL**.

7
Protocols - EFI Loaded Image

This chapter defines the **EFI_LOADED_IMAGE** protocol. This protocol describes an EFI Image that has been loaded into memory. This description includes the source from which the image was loaded, the current location of the image in memory, the type of memory allocated for the image, and the parameters passed to the image when it was invoked.

EFI_LOADED_IMAGE Protocol

Summary

Can be used on any image handle to obtain information about the loaded image.

GUID

```
#define LOADED_IMAGE_PROTOCOL        \
  {0x5B1B31A1,0x9562,0x11d2,0x8E,0x3F,0x00,0xA0,0xC9,0x69,0x72,0x3B}
```

Revision Number

```
#define EFI_LOADED_IMAGE_INFORMATION_REVISION    0x1000
```

Protocol Interface Structure

```
typedef struct {
  UINT32                    Revision;
  EFI_HANDLE                ParentHandle;
  EFI_SYSTEM_TABLE          *SystemTable;

  // Source location of the image
  EFI_HANDLE                DeviceHandle;
  EFI_DEVICE_PATH           *FilePath;
  VOID                      *Reserved;

  // Image's load options
  UINT32                    LoadOptionsSize;
  VOID                      *LoadOptions;
  // Location where image was loaded
  VOID                        *ImageBase;
  UINT64                      ImageSize;
  EFI_MEMORY_TYPE             ImageCodeType;
  EFI_MEMORY_TYPE             ImageDataType;

  EFI_IMAGE_UNLOAD             Unload;
} EFI_LOADED_IMAGE;
```

Parameters

Revision	Defines the revision of the **EFI_LOADED_IMAGE** structure. All future revisions will be backward compatible to the current revision.
ParentHandle	Parent image's image handle. **NULL** if the image is loaded directly from the firmware's boot manager. Type **EFI_HANDLE** is defined in Chapter 5.
SystemTable	The image's EFI system table pointer. Type **EFI_SYSTEM_TABLE** is defined in Chapter 4.
DeviceHandle	The device handle that the EFI Image was loaded from. Type **EFI_HANDLE** is defined in Chapter 5.
FilePath	A pointer to the file path portion specific to *DeviceHandle* that the EFI Image was loaded from. The **EFI_DEVICE_PATH** protocol is defined in Chapter 8.
Reserved	Reserved. DO NOT USE.
LoadOptionsSize	The size in bytes of *LoadOptions*.
LoadOptions	A pointer to the image's binary load options.
ImageBase	The base address at which the image was loaded.
ImageSize	The size in bytes of the loaded image.
ImageCodeType	The memory type that the code sections were loaded as. Type **EFI_MEMORY_TYPE** is defined in Chapter 5.
ImageDataType	The memory type that the data sections were loaded as. Type **EFI_MEMORY_TYPE** is defined in Chapter 5.
Unload	Function that unloads the image. See **Unload()**.

Description

Each loaded image has an image handle that supports the **EFI_LOADED_IMAGE** protocol. When an image is started, it is passed the image handle for itself. The image can use the handle to obtain its relevant image data stored in the **EFI_LOADED_IMAGE** structure, such as its load options.

LOADED_IMAGE.Unload()

Summary

Unloads an image from memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_UNLOAD_IMAGE) (
  IN EFI_HANDLE     ImageHandle,
  );
```

Parameters

ImageHandle	The handle to the image to unload. Type **EFI_HANDLE** is defined in Chapter 5.

Description

The **Unload()** function unloads an image from memory if *ImageHandle* is valid.

Status Codes Returned

EFI_SUCCESS	The image was unloaded.
EFI_INVALID_PARAMETER	The *ImageHandle* was not valid.

8
Protocols - Device Path Protocol

This chapter contains the definition of the device path protocol and the information needed to construct and manage device paths in the EFI environment. A device path is constructed and used by the firmware to convey the location of important devices, such as the boot device and console, consistent with the software-visible topology of the system.

8.1 Device Path Overview

A *Device Path* is used to define the programmatic path to a device. The primary purpose of a Device Path is to allow an application, such as an OS loader, to determine the physical device that the EFI interfaces are abstracting.

A collection of device paths is usually referred to as a name space. ACPI, for example, is rooted around a name space that is written in ASL (ACPI Source Language). Given that EFI does not replace ACPI and defers to ACPI when ever possible, it would seem logical to utilize the ACPI name space in EFI. However, the ACPI name space was designed for usage at operating system runtime and does not fit well in platform firmware or OS loaders. Given this, EFI defines its own name space, called a *Device Path*.

A Device Path is designed to make maximum leverage of the ACPI name space. One of the key structures in the Device Path defines the linkage back to the ACPI name space. The Device Path also is used to fill in the gaps where ACPI defers to buses with standard enumeration algorithms. The Device Path is able to relate information about which device is being used on buses with standard enumeration mechanisms. The Device Path is also used to define the location on a medium where a file should be, or where it was loaded from. A special case of the Device Path can also be used to support the optional booting of legacy operating systems from legacy media.

The Device Path was designed so that the OS loader and the operating system could tell which devices the platform firmware was using as boot devices. This allows the operating system to maintain a view of the system that is consistent with the platform firmware. An example of this is a "headless" system that is using a network connection as the boot device and console. In such a case, the firmware will convey to the operating system the network adapter and network protocol information being used as the console and boot device in the device path for these devices.

8.2 EFI_DEVICE_PATH Protocol

This section provides a detailed description of the **`EFI_DEVICE_PATH`** protocol.

EFI_DEVICE_PATH Protocol

Summary

Can be used on any device handle to obtain generic path/location information concerning the physical device or logical device. If the handle does not logically map to a physical device, the handle may not necessarily support the device path protocol.

GUID

```
#define DEVICE_PATH_PROTOCOL          \
     { 09576e91-6d3f-11d2-8e39-00a0c969723b }
```

Protocol Interface Structure

```
EFI_DEVICE_PATH       *DevicePath ;
```

Parameters

DevicePath	A pointer to device path data. The device path describes the location of the device the handle is for. The size of the Device Path can be determined from the structures that make up the Device Path. Type **`EFI_DEVICE_PATH`** is defined in the **`LocateDevicePath()`** function description.

Description

The executing EFI Image may use the device path to match its own device drivers to the particular device. Note that the executing EFI OS loader and EFI application images must access all physical devices via Boot Services device handles until **`ExitBootServices()`** is successfully called. An EFI driver may access only a physical device for which it provides functionality.

8.3 Device Path Nodes

There are six major types of Device Path nodes:

- Hardware Device Path. This Device Path defines how a device is attached to the resource domain of a system, where resource domain is simply the shared memory, memory mapped I/O, and I/O space of the system.
- ACPI Device Path. This Device Path is used to describe devices whose enumeration is not described in an industry-standard fashion. These devices must be described using ACPI AML in the ACPI name space; this Device Path is a linkage to the ACPI name space.
- Messaging Device Path. This Device Path is used to describe the connection of devices outside the resource domain of the system. This Device Path can describe physical messaging information (e.g., a SCSI ID) or abstract information (e.g., networking protocol IP addresses).
- Media Device Path. This Device Path is used to describe the portion of a medium that is being abstracted by a boot service. For example, a Media Device Path could define which partition on a hard drive was being used.
- BIOS Boot Specification Device Path. This Device Path is used to point to boot legacy operating systems; it is based on the BIOS Boot Specification Version 1.01. Refer to the References appendix for details on obtaining this specification.
- End of Hardware Device Path. Depending on the Sub-Type, this Device Path node is used to indicate the end of the Device Path instance or Device Path structure.

8.3.1 Generic Device Path Structures

A Device Path is a variable-length binary structure that is made up of variable-length generic Device Path nodes. Table 8-1 defines the structure of a such a node and the lengths of its components. The table defines the type and sub-type values corresponding to the Device Paths described Section 8.3; all other type and sub-type values are *Reserved*.

Table 8-1. Generic Device Path Node Structure

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 0x01 – Hardware Device Path Type 0x02 – ACPI Device Path Type 0x03 – Messaging Device Path Type 0x04 – Media Device Path Type 0x05 – BIOS Boot Specification Device Path Type 0xFF – End of Hardware Device Path
Sub-Type	1	1	Sub-Type – Varies by Type. (See Table 8-2.)
Length	2	2	Length of this structure in bytes. Length is 4 + *n* bytes.
Specific Device Path Data	4	*n*	Specific Device Path data. Type and Sub-Type define type of data. Size of data is included in Length.

A Device Path is a series of generic Device Path nodes. The first Device Path node starts at byte offset zero of the Device Path. The next Device Path node starts at the end of the previous Device Path node. Therefore all nodes are byte-packed data structures that may appear on any byte boundary. All code references to device path notes must assume all fields are **`UNALIGNED`**. Since every Device Path node contains a length field in a known place, it is possible to traverse Device Path nodes that are of an unknown type. There is no limit to the number, type, or sequence of nodes in a Device Path.

A Device Path is terminated by an End of Hardware Device Path node. This type of node has two sub-types (see Table 8-2):

- *End This Instance of a Device Path* (sub-type 0x01). This type of node terminates one Device Path instance and denotes the start of another. This is only required when an environment variable represents multiple devices. An example of this would be the **`ConsoleOut`** environment variable that consists of both a VGA console and serial output console. This variable would describe a console output stream that is sent to both VGA and serial concurrently and thus has a Device Path that contains two complete Device Paths.
- *End Entire Device Path* (sub-type 0xFF). This type of node terminates an entire Device Path. Software searches for this sub-type to find the end of a Device Path. All Device Paths must end with this sub-type.

Table 8-2. Device Path End Structure

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 0x7F – End of Hardware Device Path Type 0xFF – End of Hardware Device Path
Sub-Type	1	1	Sub-Type 0xFF – End Entire Device Path, or Sub-Type 0x01 – End This Instance of a Device Path and start a new Device Path
Length	2	2	Length of this structure in bytes. Length is 4 bytes.

8.3.2 Hardware Device Path

This Device Path defines how a device is attached to the resource domain of a system, where resource domain is simply the shared memory, memory mapped I/O, and I/O space of the system. It is possible to have multiple levels of Hardware Device Path such as a PCCARD device that was attached to a PCCARD PCI controller.

8.3.2.1 PCI Device Path

The Device Path for PCI defines the path to the PCI configuration space address for a PCI device. There is one PCI Device Path entry for each device and function number that defines the path from the root PCI bus to the device. Because the PCI bus number of a device may potentially change, a flat encoding of single PCI Device Path entry cannot be used. An example of this is when a PCI device is behind a bridge, and one of the following events occurs:

- OS performs a Plug and Play configuration of the PCI bus.
- A hot plug of a PCI device is performed.
- The system configuration changes between reboots.

The PCI Device Path entry must be preceded by an ACPI Device Path entry that uniquely identifies the PCI root bus. The programming of root PCI bridges is not defined by any PCI specification and this is why an ACPI Device Path entry is required.

Table 8-3. PCI Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 1 – Hardware Device Path
Sub-Type	1	1	Sub-Type 1 – PCI
Length	2	2	Length of this structure is 6 bytes
Function	4	1	PCI Function Number
Device	5	1	PCI Device Number

8.3.2.2 PCCARD Device Path

Table 8-4. PCCARD Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 1 – Hardware Device Path
Sub-Type	1	1	Sub-Type 2 – PCCARD
Length	2	2	Length of this structure in bytes. Length is 5 bytes.
Function Number	4	1	Function Number (0 = First Function)

8.3.2.3 Memory Mapped Device Path

Table 8-5. Memory Mapped Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 1 – Hardware Device Path.
Sub-Type	1	1	Sub-Type 3 – Memory Mapped.
Length	2	2	Length of this structure in bytes. Length is 24 bytes.
Memory Type	4	4	**`EFI_MEMORY_TYPE`**. Type **`EFI_MEMORY_TYPE`** is defined in the **`AllocatePages()`** function description.
Start Address	8	8	Starting Memory Address.
End Address	16	8	Ending Memory Address.

8.3.2.4 Vendor Device Path

The Vendor Device Path allows the creation of vendor-defined Device Paths. A vendor must allocate a Vendor_GUID for a Device Path. The Vendor_GUID can then be used to define the contents on the *n* bytes that follow in the Vendor Device Path node.

Table 8-6. Vendor-Defined Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 1 – Hardware Device Path.
Sub-Type	1	1	Sub-Type 4 – Vendor.
Length	2	2	Length of this structure in bytes. Length is 20 + *n* bytes.
Vendor_GUID	4	16	Vendor-assigned GUID that defines the data that follows.
Vendor Defined Data	20	*n*	Vendor-defined variable size data.

8.3.2.5 Controller Device Path

Table 8-7. Controller Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 1 – Hardware Device Path.
Sub-Type	1	1	Sub-Type 5 – Controller.
Length	2	2	Length of this structure in bytes. Length is 8 bytes.
Controller Number	4	4	Controller number.

8.3.3 ACPI Device Path

This Device Path contains ACPI Device IDs that represent a device's Plug and Play Hardware ID and its corresponding unique persistent ID. The ACPI IDs are stored in the ACPI _HID, _CID, and _UID device identification objects that are associated with a device. The ACPI Device Path contains values that must match exactly the ACPI name space that is provided by the platform firmware to the operating system. Refer to the ACPI specification for a complete description of the _HID, _CID, and _UID device identification objects.

The _HID and _CID values are optional device identification objects that appear in the ACPI name space. If only _HID is present, the _HID must be used to describe any device that will be enumerated by the ACPI driver. The _CID, if present, contains information that is important for the OS to attach generic driver (e.g., PCI Bus Driver), while the _HID contains information important for the OS to attach device-specific driver. The ACPI bus driver only enumerates a device when no standard bus enumerator exists for a device.

The _UID object provides the OS with a serial number-style ID for a device that does not change across reboots. The object is optional, but is required when a system contains two devices that report the same _HID. The _UID only needs to be unique among all device objects with the same _HID value. If no _UID exists in the APCI name space for a _HID the value of zero must be stored in the _UID field of the ACPI Device Path.

The ACPI Device Path is only used to describe devices that are not defined by a Hardware Device Path. An _HID (along with _CID if present) is required to represent a PCI root bridge, since the PCI specification does not define the programming model for a PCI root bridge. There are two subtypes of the ACPI Device Path: a simple subtype that only includes the _HID and _UID fields, and an extended subtype that includes the _HID, _CID, and _UID fields.

The ACPI Device Path node only supports numeric 32-bit values for the _HID and _UID values. The Expanded ACPI Device Path node supports both numeric and string values for the _HID, _UID, and _CID values. As a result, the ACPI Device Path node is smaller and should be used if possible to reduce the size of device paths that may potentially be stored in nonvolatile storage. If a string value is required for the _HID field, or a string value is required for the _UID field, or a _CID field is required, then the Expanded ACPI Device Path node must be used. If a string field of the Expanded ACPI Device Path node is present, then the corresponding numeric field is ignored.

The _HID and _CID fields in the ACPI Device Path node and Expanded ACPI Device Path node are stored as a 32-bit compressed EISA-type IDs. The following macro can be used to compute these EISA-type IDs from a Plug and Play Hardware ID. The Plug and Play Hardware IDs used to compute the _HID and _CID fields in the EFI device path nodes must match the Plug and Play Hardware IDs used to build the matching entries in the ACPI tables. The compressed EISA-type IDs produced by this macro differ from the compressed EISA-type IDs stored in ACPI tables. As a result, the compressed EISA-type IDs from the ACPI Device Path nodes cannot be directly compared to the compressed EISA-type IDs from the ACPI table.

```
#define EFI_PNP_ID(ID)  (UINT32)(((ID) << 16) | 0x41D0)
#define EISA_PNP_ID(ID) EFI_PNP_ID(ID)
```

Table 8-8. ACPI Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 2 – ACPI Device Path.
Sub-Type	1	1	Sub-Type 1 ACPI Device Path.
Length	2	2	Length of this structure in bytes. Length is 12 bytes.
_HID	4	4	Device's PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID. This value must match the corresponding _HID in the ACPI name space.
_UID	8	4	Unique ID that is required by ACPI if two devices have the same _HID. This value must also match the corresponding _UID/_HID pair in the ACPI name space. Only the 32-bit numeric value type of _UID is supported; thus strings must not be used for the _UID in the ACPI name space.

Table 8-9. Expanded ACPI Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 2 – ACPI Device Path.
Sub-Type	1	1	Sub-Type 2 Expanded ACPI Device Path.
Length	2	2	Length of this structure in bytes. Minimum length is 19 bytes. The actual size will depend on the size of the _HIDSTR, _UIDSTR, and _CIDSTR fields.
_HID	4	4	Device's PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID. This value must match the corresponding _HID in the ACPI name space.
_UID	8	4	Unique ID that is required by ACPI if two devices have the same _HID. This value must also match the corresponding _UID/_HID pair in the ACPI name space.
_CID	12	4	Device's compatible PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID. This value must match at least one of the compatible device IDs returned by the corresponding _CID in the ACPI name space.
_HIDSTR	16	>=1	Device's PnP hardware ID stored as a null-terminated ASCII string. This value must match the corresponding _HID in the ACPI name space. If the length of this string not including the null-terminator is 0, then the _HID field is used. If the length of this null-terminated string is greater than 0, then this field supercedes the _HID field.

continued

Table 8-9. Expanded ACPI Device Path (continued)

Mnemonic	Byte Offset	Byte Length	Description
_UIDSTR	Varies	>=1	Unique ID that is required by ACPI if two devices have the same _HID. This value must also match the corresponding _UID/_HID pair in the ACPI name space. This value is stored as a null-terminated ASCII string. If the length of this string not including the null-terminator is 0, then the _UID field is used. If the length of this null-terminated string is greater than 0, then this field supercedes the _UID field. The Byte Offset of this field can be computed by adding 16 to the size of the _HIDSTR field.
_CIDSTR	Varies	>=1	Device's compatible PnP hardware ID stored as a null-terminated ASCII string. This value must match at least one of the compatible device IDs returned by the corresponding _CID in the ACPI name space. If the length of this string not including the null-terminator is 0, then the _CID field is used. If the length of this null-terminated string is greater than 0, then this field supercedes the _CID field. The Byte Offset of this field can be computed by adding 16 to the sum of the sizes of the _HIDSTR and _UIDSTR fields.

8.3.4 Messaging Device Path

This Device Path is used to describe the connection of devices outside the resource domain of the system. This Device Path can describe physical messaging information like SCSI ID or abstract information like networking protocol IP addresses.

8.3.4.1 ATAPI Device Path

Table 8-10. ATAPI Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 1 – ATAPI
Length	2	2	Length of this structure in bytes. Length is 8 bytes.
PrimarySecondary	4	1	Set to zero for primary or one for secondary
SlaveMaster	5	1	Set to zero for master or one for slave mode
Logical Unit Number	6	2	Logical Unit Number

8.3.4.2 SCSI Device Path

Table 8-11. SCSI Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 2 – SCSI
Length	2	2	Length of this structure in bytes. Length is 8 bytes.
Target ID	4	2	Target ID on the SCSI bus, PUN
Logical Unit Number	6	2	Logical Unit Number, LUN

8.3.4.3 Fibre Channel Device Path

Table 8-12. Fibre Channel Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 3 – Fibre Channel
Length	2	2	Length of this structure in bytes. Length is 24 bytes.
Reserved	4	4	Reserved
World Wide Number	8	8	Fibre Channel World Wide Number
Logical Unit Number	16	8	Fibre Channel Logical Unit Number

8.3.4.4 1394 Device Path

Table 8-13. 1394 Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 4 – 1394
Length	2	2	Length of this structure in bytes. Length is 16 bytes.
Reserved	4	4	Reserved
GUID[1]	8	8	1394 Global Unique ID (GUID)[1]
Notes: [1]The usage of the term GUID is per the 1394 specification. This is not the same as the **`EFI_GUID`** type defined in the EFI Specification.			

8.3.4.5 USB Device Path

Table 8-14. USB Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 5 – USB
Length	2	2	Length of this structure in bytes. Length is 16 bytes.
USB Parent Port Number	4	1	USB Parent Port Number
Interface	5	1	USB Interface Number

8.3.4.6 USB Class Device Path

Table 8-15. USB Class Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 - Messaging Device Path.
Sub-Type	1	1	Sub-Type 15 - USB Class.
Length	2	2	Length of this structure in bytes. Length is 11 bytes.
Vendor ID	4	2	Vendor ID assigned by USB-IF. A value of 0xFFFF will match any Vendor ID.
Product ID	6	2	Product ID assigned by USB-IF. A value of 0xFFFF will match any Product ID.
Device Class	8	1	The class code assigned by the USB-IF. A value of 0xFF will match any class code.
Device Subclass	9	1	The subclass code assigned by the USB-IF. A value of 0xFF will match any subclass code.
Device Protocol	10	1	The protocol code assigned by the USB-IF. A value of 0xFF will match any protocol code.

8.3.4.7 I_2O Device Path

Table 8-16. I_2O Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 6 – I2O Random Block Storage Class
Length	2	2	Length of this structure in bytes. Length is 8 bytes.
TID	4	4	Target ID (TID) for a device

8.3.4.8 MAC Address Device Path

Table 8-17. MAC Address Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 11 – MAC Address for a network interface
Length	2	2	Length of this structure in bytes. Length is 37 bytes.
MAC Address	4	32	The MAC address for a network interface padded with 0s
IfType	36	1	Network interface type(i.e. 802.3, FDDI). See RFC 1700

8.3.4.9 IPv4 Device Path

Table 8-18. IPv4 Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 12 – IPv4
Length	2	2	Length of this structure in bytes. Length is 19 bytes.
Local IP Address	4	4	The local IPv4 address
Remote IP Address	8	4	The remote IPv4 address
Local Port	12	2	The local port number
Remote Port	14	2	The remote port number
Protocol	16	2	The network protocol(i.e. UDP, TCP). See RFC 1700
StaticIPAddress	18	1	0x00 - The Source IP Address was assigned though DHCP 0x01 - The Source IP Address is statically bound

8.3.4.10 IPv6 Device Path

Table 8-19. IPv6 Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 13 – IPv6
Length	2	2	Length of this structure in bytes. Length is 43 bytes.
Local IP Address	4	16	The local IPv6 address
Remote IP Address	20	16	The remote IPv6 address
Local Port	36	2	The local port number
Remote Port	38	2	The remote port number
Protocol	40	2	The network protocol (i.e. UDP, TCP). See RFC 1700
StaticIPAddress	42	1	0x00 - The Source IP Address was assigned though DHCP 0x01 - The Source IP Address is statically bound

8.3.4.11 InfiniBand Device Path

Table 8-20. InfiniBand Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 9 – InfiniBand
Length	2	2	Length of this structure in bytes. Length is 48 bytes.
Resource Flags	4	4	Flags to help identify/manage InfiniBand device path elements: • Bit 0 – IOC/Service (0b = IOC, 1b = Service) • Bit 1 – Extend Boot Environment • Bit 2 – Console Protocol • Bit 3 – Storage Protocol • Bit 4 – Network Protocol All other bits are reserved.
PORT GID	8	16	128-bit Global Identifier for remote fabric port
IOC GUID/Service ID	24	8	64-bit unique identifier to remote IOC or server process. Interpretation of field specified by Resource Flags (bit 0)
Target Port ID	32	8	64-bit persistent ID of remote IOC port
Device ID	40	8	64-bit persistent ID of remote device

Note: The usage of the terms GUID and GID is per the InfiniBand Specification. The term GUID is not the same as the `EFI_GUID` type defined in this EFI Specification.

8.3.4.12 UART Device Path

Table 8-21. UART Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 14 – UART
Length	2	2	Length of this structure in bytes. Length is 19 bytes.
Reserved	4	4	Reserved
Baud Rate	8	8	The baud rate setting for the UART style device. A value of 0 means that the device's default baud rate will be used.
Data Bits	16	1	The number of data bits for the UART style device. A value of 0 means that the device's default number of data bits will be used.
Parity	17	1	The parity setting for the UART style device. Parity 0x00 - Default Parity Parity 0x01 - No Parity Parity 0x02 - Even Parity Parity 0x03 - Odd Parity Parity 0x04 - Mark Parity Parity 0x05 - Space Parity
Stop Bits	18	1	The number of stop bits for the UART style device. Stop Bits 0x00 - Default Stop Bits Stop Bits 0x01 - 1 Stop Bit Stop Bits 0x02 - 1.5 Stop Bits Stop Bits 0x03 - 2 Stop Bits

8.3.4.13 Vendor-Defined Messaging Device Path

Table 8-22. Vendor-Defined Messaging Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 10 – Vendor
Length	2	2	Length of this structure in bytes. Length is 20 + *n* bytes.
Vendor_GUID	4	16	Vendor-assigned GUID that defines the data that follows
Vendor Defined Data	20	*n*	Vendor-defined variable size data

The following two GUIDs are used with a Vendor-Defined Messaging Device Path to describe the transport protocol for use with PC-ANSI, VT-100, VT-100+, and VT-UTF8 terminals. Device paths can be constructed with this node as the last node in the device path. The rest of the device path describes the physical device that is being used to transmit and receive data. The PC-ANSI, VT-100, VT-100+, and VT-UTF8 GUIDs define the format of the data that is being sent though the physical device. Additional GUIDs can be generated to describe additional transport protocols.

```
#define EFI_PC_ANSI_GUID  \
  { 0xe0c14753,0xf9be,0x11d2,0x9a,0x0c,0x00,0x90,0x27,0x3f,0xc1,0x4d }

#define EFI_VT_100_GUID  \
  { 0xdfa66065,0xb419,0x11d3,0x9a,0x2d,0x00,0x90,0x27,0x3f,0xc1,0x4d }

#define EFI_VT_100_PLUS_GUID  \
  { 0x7baec70b,0x57e0,0x4c76,0x8e,0x87,0x2f,0x9e,0x28,0x08,0x83,0x43 }

#define EFI_VT_UTF8_GUID  \
  { 0xad15a0d6,0x8bec,0x4acf,0xa0,0x73,0xd0,0x1d,0xe7,0x7e,0x2d,0x88 }
```

8.3.4.14 UART Flow Control Messaging Path

The UART messaging device path defined in the EFI 1.02 specification does not contain a provision for flow control. Therefore, a new device path node is needed to declare flow control characteristics. It is a vendor-defined messaging node which may be appended to the UART node in a device path. It has the following definition:

```
#define DEVICE_PATH_MESSAGING_UART_FLOW_CONTROL    \
{0X37499A9D,0X542F,0X4C89,0XA0,0X26,0X35,0XDA,0X14,0X20,0X94,0XE4}
```

Table 8-23. UART Flow Control Messaging Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 10 – Vendor
Length	2	2	Length of this structure in bytes. Length is 24 bytes.
Vendor_GUID	4	16	`DEVICE_PATH_MESSAGING_UART_FLOW_CONTROL`
Flow_Control_Map	20	4	Bitmap of supported flow control types. Bit 0 set indicates hardware flow control. Bit 1 set indicates Xon/Xoff flow control. All other bits are reserved and are clear.

A debugport driver that implements Xon/Xoff flow control would produce a device path similar to the following:

```
ACPI(PciRootBridge)/Pci(0x1f,0)/ACPI(PNP0501,0)/UART(115200,n,8,1)
/UartFlowCtrl(2)/DebugPort()
```

NOTE

If no bits are set in the Flow_Control_Map, this indicates there is no flow control and is equivalent to leaving the flow control node out of the device path completely.

8.3.5 Media Device Path

This Device Path is used to describe the portion of the medium that is being abstracted by a boot service. An example of Media Device Path would be defining which partition on a hard drive was being used.

8.3.5.1 Hard Drive

The Hard Drive Media Device Path is used to represent a partition on a hard drive. The master boot record (MBR) that resides in the first sector of the disk defines the partitions on a disk. Partitions are addressed in EFI starting at LBA zero. Partitions are numbered one through *n*. A partition number of zero can be used to represent the raw hard drive.

The MBR Type is stored in the Device Path to allow new MBR types to be added in the future. The Hard Drive Device Path also contains a Disk Signature and a Disk Signature Type. The disk signature is maintained by the OS and only used by EFI to partition Device Path nodes. The disk signature enables the OS to find disks even after they have been physically moved in a system.

Table 8-24. Hard Drive Media Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 4 – Media Device Path
Sub-Type	1	1	Sub-Type 1 – Hard Drive
Length	2	2	Length of this structure in bytes. Length is 42 bytes.
Partition Number	4	4	Partition Number of the hard drive. Partition numbers start at one. Partition number zero represents the entire device. Partitions are defined by entries in the master boot record in the first sector of the hard disk device.
Partition Start	8	8	Starting LBA of the partition on the hard drive
Partition Size	16	8	Size of the partition in units of Logical Blocks
Partition Signature	24	16	Signature unique to this partition
MBR Type	40	1	MBR Type: (Unused values reserved) 0x01 – PC-AT compatible MBR. Partition Start and Partition Size come from `PartitionStartingLBA` and `PartitionSizeInLBA` for the partition. 0x02 – EFI Partition Table Header.
Signature Type	41	1	Type of Disk Signature: (Unused values reserved) 0x00 – No Disk Signature. 0x01 – 32-bit signature from address 0x1b8 of the type 0x01 MBR. 0x02 – GUID signature.

The following structure defines a MBR for EFI:

```
typedef struct _MBR_PARTITION {
       UINT8         BootIndicator;      // 0x80 for active partition
       UINT8         PartitionStartCHS[3];
       UINT8         OS_Indicator;
       UINT8         PartitionEndCHS[3];
       UINT32        PartitionStartingLBA;
       UINT32        PartitionSizeInLBA;
} MBR_PARTITION;

typedef struct _PC_MBR {
       UINT8                MBRCode[0x1BE];
       MBR_PARTITION        PartitionEntry[4];
       UINT16               Signature;          // Must be 0xaa55
} PC_MBR;
```

8.3.5.2 CD-ROM Media Device Path

The CD-ROM Media Device Path is used to define a system partition that exists on a CD-ROM. The CD-ROM is assumed to contain an ISO-9660 file system and follow the CD-ROM "El Torito" format. The Boot Entry number from the Boot Catalog is how the "El Torito" specification defines the existence of bootable entities on a CD-ROM. In EFI the bootable entity is an EFI System Partition that is pointed to by the Boot Entry.

Table 8-25. CD-ROM Media Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 4 – Media Device Path.
Sub-Type	1	1	Sub-Type 2 – CD-ROM "El Torito" Format.
Length	2	2	Length of this structure in bytes. Length is 24 bytes.
Boot Entry	4	4	Boot Entry number from the Boot Catalog. The Initial/Default entry is defined as zero.
Partition Start	8	8	Starting RBA of the partition on the medium. CD-ROMs use Relative logical Block Addressing.
Partition Size	16	8	Size of the partition in units of Blocks, also called Sectors.

8.3.5.3 Vendor-Defined Media Device Path

Table 8-26. Vendor-Defined Media Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 4 – Media Device Path.
Sub-Type	1	1	Sub-Type 3 – Vendor.
Length	2	2	Length of this structure in bytes. Length is 20 + *n* bytes.
Vendor_GUID	4	16	Vendor-assigned GUID that defines the data that follows.
Vendor Defined Data	20	*n*	Vendor-defined variable size data.

8.3.5.4 File Path Media Device Path

Table 8-27. File Path Media Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 4 – Media Device Path.
Sub-Type	1	1	Sub-Type 4 – File Path.
Length	2	2	Length of this structure in bytes. Length is 4 + *n* bytes.
Path Name	4	*n*	Unicode Path string including directory and file names. The length of this string *n* can be determined by subtracting 4 from the Length entry. A device path may contain one or more of these nodes. The complete path to a file can be found by concatenating all the File Path Media Device Path nodes. This is typically used to describe the directory path in one node, and the filename in another node.

8.3.5.5 Media Protocol Device Path

The Media Protocol Device Path is used to denote the protocol that is being used in a device path at the location of the path specified. Many protocols are inherent to the style of device path.

Table 8-28. Media Protocol Media Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 4 – Media Device Path.
Sub-Type	1	1	Sub-Type 5 – Media Protocol.
Length	2	2	Length of this structure in bytes. Length is 20 bytes.
Protocol GUID	4	16	The ID of the protocol.

8.3.6 BIOS Boot Specification Device Path

This Device Path is used to describe the booting of non-EFI-aware operating systems. This Device Path is based on the IPL and BCV table entry data structures defined in Appendix A of the *BIOS Boot Specification*. The BIOS Boot Specification Device Path defines a complete Device Path and is not used with other Device Path entries. This Device Path is only needed to enable platform firmware to select a legacy non-EFI OS as a boot option.

Table 8-29. BIOS Boot Specification Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 5 – BIOS Boot Specification Device Path.
Sub-Type	1	1	Sub-Type 1 – BIOS Boot Specification Version 1.01.
Length	2	2	Length of this structure in bytes. Length is 8 + *n* bytes.
Device Type	4	2	Device Type as defined by the BIOS Boot Specification.
Status Flag	6	2	Status Flags as defined by the BIOS Boot Specification
Description String	8	*n*	ASCIIZ string that describes the boot device to a user. The length of this string *n* can be determined by subtracting 8 from the Length entry.

Example BIOS Boot Specification Device Types would include:

- 00h = Reserved
- 01h = Floppy
- 02h = Hard Disk
- 03h = CD-ROM
- 04h = PCMCIA
- 05h = USB
- 06h = Embedded network
- 07h..7Fh = Reserved
- 80h = BEV device
- 81h..FEh = Reserved
- FFh = Unknown

8.4 Device Path Generation Rules

8.4.1 Housekeeping Rules

The Device Path is a set of Device Path nodes. The Device Path must be terminated by an End of Device Path node with a sub-type of End the Entire Device Path. A NULL Device Path consists of a single End Device Path Node. A Device Path that contains a NULL pointer and no Device Path structures is illegal.

All Device Path nodes start with the generic Device Path structure. Unknown Device Path types can be skipped when parsing the Device Path since the length field can be used to find the next Device Path structure in the stream. Any future additions to the Device Path structure types will always start with the current standard header. The size of a Device Path can be determined by traversing the generic Device Path structures in each header and adding up the total size of the Device Path. This size will include the four bytes of the End of Device Path structure.

Multiple hardware devices may be pointed to by a single Device Path. Each hardware device will contain a complete Device Path that is terminated by the Device Path End Structure. The Device Path End Structures that do not end the Device Path contain a sub-type of End This Instance of the Device Path. The last Device Path End Structure contains a sub-type of End Entire Device Path.

8.4.2 Rules with ACPI _HID and _UID

As described in the ACPI specification, ACPI supports several different kinds of device identification objects, including _HID, _CID and _UID. The _UID device identification objects are optional in ACPI and only required if more than one _HID exists with the same ID. The ACPI Device Path structure must contain a zero in the _UID field if the ACPI name space does not implement _UID. The _UID field is a unique serial number that persists across reboots.

If a device in the ACPI name space has a _HID and is described by a _CRS (Current Resource Setting) then it should be described by an ACPI Device Path structure. A _CRS implies that a device is not mapped by any other standard. A _CRS is used by ACPI to make a nonstandard device into a Plug and Play device. The configuration methods in the ACPI name space allow the ACPI driver to configure the device in a standard fashion. The presence of a _CID determines whether the ACPI Device Path node or the Expanded ACPI Device Path node should be used.

Table 8-30 maps ACPI _CRS devices to EFI Device Path.

Table 8-30. ACPI _CRS to EFI Device Path Mapping

ACPI _CRS Item	EFI Device Path
PCI Root Bus	ACPI Device Path: _HID PNP0A03, _UID
Floppy	ACPI Device Path: _HID PNP0604, _UID drive select encoding 0-3
Keyboard	ACPI Device Path: _HID PNP0301, _UID 0
Serial Port	ACPI Device Path: _HID PNP0501, _UID Serial Port COM number 0-3
Parallel Port	ACPI Device Path: _HID PNP0401, _UID LPT number 0-3

Support of root PCI bridges requires special rules in the EFI Device Path. A root PCI bridge is a PCI device usually contained in a chipset that consumes a proprietary bus and produces a PCI bus. In typical desktop and mobile systems there is only one root PCI bridge. On larger server systems there are typically multiple root PCI bridges. The operation of root PCI bridges is not defined in any current PCI specification. A root PCI bridge should not be confused with a PCI to PCI bridge that both consumes and produces a PCI bus. The operation and configuration of PCI to PCI bridges is fully specified in current PCI specifications.

Root PCI bridges will use the plug and play ID of PNP0A03, This will be stored in the ACPI Device Path _HID field, or in the Expanded ACPI Device Path _CID field to match the ACPI name space. The _UID in the ACPI Device Path structure must match the _UID in the ACPI name space.

8.4.3 Rules with ACPI _ADR

If a device in the ACPI name space can be completely described by a _ADR object then it will map to an EFI ACPI, Hardware, or Message Device Path structure. A _ADR method implies a bus with a standard enumeration algorithm. If the ACPI device has a _ADR and a _CRS method, then it should also have a _HID method and follow the rules for using _HID.

Table 8-31 relates the ACPI_ADR bus definition to the EFI Device Path:

Table 8-31. ACPI _ADR to EFI Device Path Mapping

ACPI _ADR Bus	EFI Device Path
EISA	*Not supported*
Floppy Bus	ACPI Device Path: _HID PNP0604, _UID drive select encoding 0-3
IDE Controller	ATAPI Message Device Path: Maser/Slave : LUN
IDE Channel	ATAPI Message Device Path: Maser/Slave : LUN
PCI	PCI Hardware Device Path
PCMCIA	*Not Supported*
PC CARD	PC CARD Hardware Device Path
SMBus	*Not Supported*

8.4.4 Hardware vs. Messaging Device Path Rules

Hardware Device Paths are used to define paths on buses that have a standard enumeration algorithm and that relate directly to the coherency domain of the system. The coherency domain is defined as a global set of resources that is visible to at least one processor in the system. In a typical system this would include the processor memory space, IO space, and PCI configuration space.

Messaging Device Paths are used to define paths on buses that have a standard enumeration algorithm, but are not part of the global coherency domain of the system. SCSI and Fibre Channel are examples of this kind of bus. The Messaging Device Path can also be used to describe virtual connections over network-style devices. An example would be the TCPI/IP address of a internet connection.

Thus Hardware Device Path is used if the bus produces resources that show up in the coherency resource domain of the system. A Message Device Path is used if the bus consumes resources from the coherency domain and produces resources out side the coherency domain of the system.

8.4.5 Media Device Path Rules

The Media Device Path is used to define the location of information on a medium. Hard Drives are subdivided into partitions by the MBR and a Media Device Path is used to define which partition is being used. A CD-ROM has boot partitions that are defined by the "El Torito" specification, and the Media Device Path is used to point to these partitions.

A **`BLOCK_IO`** protocol is produced for both raw devices and partitions on devices. This allows the **`SIMPLE_FILE_SYSTEM`** protocol to not have to understand media formats. The **`BLOCK_IO`** protocol for a partition contains the same Device Path as the parent **`BLOCK_IO`** protocol for the raw device with the addition of a Media Device Path that defines which partition is being abstracted.

The Media Device Path is also used to define the location of a file in a file system. This Device Path is used to load files and to represent what file an image was loaded from.

8.4.6 Other Rules

The BIOS Boot Specification Device Path is not a typical Device Path. A Device Path containing the BIOS Boot Specification Device Path should only contain the required End Device Path structure and no other Device Path structures. The BIOS Boot Specification Device Path is only used to allow the EFI boot menus to boot a legacy operating system from legacy media.

The EFI Device Path can be extended in a compatible fashion by assigning your own vendor GUID to a Hardware, Messaging, or Media Device Path. This extension is guaranteed to never conflict with future extensions of this specification

The EFI specification reserves all undefined Device Path types and subtypes. Extension is only permitted using a Vendor GUID Device Path entry.

9
Protocols - EFI Driver Model

EFI drivers that follow the *EFI Driver Model* are not allowed to search for controllers to manage. When a specific controller is needed, the EFI boot service **ConnectController()** is used along with the **EFI_DRIVER_BINDING_PROTOCOL** services to identify the best drivers for a controller. Once **ConnectController()** has identified the best drivers for a controller, the start service in the **EFI_DRIVER_BINDING_PROTOCOL** is used by **ConnectController()** to start each driver on the controller. Once a controller is no longer needed, it can be released with the EFI boot service **DisconnectController()**. **DisconnectController()** calls the stop service in each **EFI_DRIVER_BINDING_PROTOCOL** to stop the controller.

The driver initialization routine of an EFI driver is not allowed to touch any device hardware. Instead, it just installs an instance of the **EFI_DRIVER_BINDING_PROTOCOL** on the *ImageHandle* of the EFI driver. The test to determine if a driver supports a given controller must be performed in as little time as possible without causing any side effects on any of the controllers it is testing. As a result, most of the controller initialization code is present in the start and stop services of the **EFI_DRIVER_BINDING_PROTOCOL**.

9.1 EFI Driver Binding Protocol

This section provides a detailed description of the **EFI_DRIVER_BINDING_PROTOCOL**. This protocol is produced by every driver that follows the *EFI Driver Model*, and it is the central component that allows drivers and controllers to be managed. It provides a service to test if a specific controller is supported by a driver, a service to start managing a controller, and a service to stop managing a controller. These services apply equally to drivers for both bus controllers and device controllers.

EFI_DRIVER_BINDING_PROTOCOL

Summary

Provides the services required to determine if a driver supports a given controller. If a controller is supported, then it also provides routines to start and stop the controller.

GUID

```
#define EFI_DRIVER_BINDING_PROTOCOL_GUID  \
{0x18A031AB,0xB443,0x4D1A,0xA5,0xC0,0x0C,0x09,0x26,0x1E,0x9F,0x71}
```

Protocol Interface Structure

```
typedef struct _EFI_DRIVER_BINDING_PROTOCOL {
  EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED  Supported;
  EFI_DRIVER_BINDING_PROTOCOL_START      Start;
  EFI_DRIVER_BINDING_PROTOCOL_STOP       Stop;
  UINT32                                 Version;
  EFI_HANDLE                             ImageHandle;
  EFI_HANDLE                             DriverBindingHandle;
} EFI_DRIVER_BINDING_PROTOCOL;
```

Parameters

Supported	Tests to see if this driver supports a given controller. This service is called by the EFI boot service **ConnectController()**. In order to make drivers as small as possible, there are a few calling restrictions for this service. **ConnectController()** must follow these calling restrictions. If any other agent wishes to call **Supported()** it must also follow these calling restrictions. See the **Supported()** function description.
Start	Starts a controller using this driver. This service is called by the EFI boot service **ConnectController()**. In order to make drivers as small as possible, there are a few calling restrictions for this service. **ConnectController()** must follow these calling restrictions. If any other agent wishes to call **Start()** it must also follow these calling restrictions. See the **Start()** function description.
Stop	Stops a controller using this driver. This service is called by the EFI boot service **DisconnectController()**. In order to make drivers as small as possible, there are a few calling restrictions for this service. **DisconnectController()** must follow these calling restrictions. If any other agent wishes to call **Stop()** it must also follow these calling restrictions. See the **Stop()** function description.
Version	The version number of the EFI Driver that produced the **EFI_DRIVER_BINDING_PROTOCOL**. This field is used by the EFI boot service **ConnectController()** to determine the order that driver's **Supported()** service will be used when a controller needs to be started. EFI Driver Binding Protocol instances with higher *Version* values will be used before ones with lower *Version* values. The *Version* values of *0x0-0x0f* and *0xfffffff0-0xffffffff* are reserved for platform/OEM specific drivers. The *Version* values of *0x10-0xffffffef* are reserved for IHV-developed drivers.

ImageHandle	The image handle of the EFI Driver that produced this instance of the **EFI_DRIVER_BINDING_PROTOCOL**.
DriverBindingHandle	The handle on which this instance of the **EFI_DRIVER_BINDING_PROTOCOL** is installed. In most cases, this is the same handle as *ImageHandle*. However, for EFI Drivers that produce more than one instance of the **EFI_DRIVER_BINDING_PROTOCOL**, this value may not be the same as *ImageHandle*.

Description

The **EFI_DRIVER_BINDING_PROTOCOL** provides a service to determine if a driver supports a given controller. If a controller is supported, then it also provides services to start and stop the controller. All EFI drivers are required to be reentrant so they can manage one or more controllers. This requires that drivers not use global variables to store device context. Instead, they must allocate a separate context structure per controller that the driver is managing. Bus drivers must support starting and stopping the same bus multiple times, and they must also support starting and stopping all of their children, or just a subset of their children.

EFI_DRIVER_BINDING_PROTOCOL.Supported()

Summary

Tests to see if this driver supports a given controller. If a child device is provided, it further tests to see if this driver supports creating a handle for the specified child device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED) (
  IN  EFI_DRIVER_BINDING_PROTOCOL   *This,
  IN  EFI_HANDLE                    ControllerHandle,
  IN  EFI_DEVICE_PATH_PROTOCOL      *RemainingDevicePath  OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_DRIVER_BINDING_PROTOCOL** instance.
ControllerHandle	The handle of the controller to test. This handle must support a protocol interface that supplies an I/O abstraction to the driver. Sometimes just the presence of this I/O abstraction is enough for the driver to determine if it supports *ControllerHandle*. Sometimes, the driver may use the services of the I/O abstraction to determine if this driver supports *ControllerHandle*.
RemainingDevicePath	A pointer to the remaining portion of a device path. This parameter is ignored by device drivers, and is optional for bus drivers. For bus drivers, if this parameter is not **NULL**, then the bus driver must determine if the bus controller specified by *ControllerHandle* and the child controller specified by *RemainingDevicePath* are both supported by this bus driver.

Description

This function checks to see if the driver specified by *This* supports the device specified by *ControllerHandle*. Drivers will typically use the device path attached to *ControllerHandle* and/or the services from the bus I/O abstraction attached to *ControllerHandle* to determine if the driver supports *ControllerHandle*. This function may be called many times during platform initialization. In order to reduce boot times, the tests performed by this function must be very small, and take as little time as possible to execute. This function must not change the state of any hardware devices, and this function must be aware that the device specified by *ControllerHandle* may already be managed by the same driver or a different driver. This function must match its calls to **AllocatePages()** with **FreePages()**, **AllocatePool()** with **FreePool()**, and **OpenProtocol()** with

CloseProtocol(). Since *ControllerHandle* may have been previously started by the same driver, if a protocol is already in the opened state, then it must not be closed with **CloseProtocol()**. This is required to guarantee the state of *ControllerHandle* is not modified by this function.

If any of the protocol interfaces on the device specified by *ControllerHandle* that are required by the driver specified by *This* are already open for exclusive access by a different driver or application, then **EFI_ACCESS_DENIED** is returned.

If any of the protocol interfaces on the device specified by *ControllerHandle* that are required by the driver specified by *This* are already opened by the same driver, then **EFI_ALREADY_STARTED** is returned. However, if the driver specified by *This* is a bus driver that is able to create one child handle at a time, then it is not an error, and the bus driver should continue with its test of *ControllerHandle*. This allows a bus driver to create one child handle on the first call to **Supported()** and **Start()**, and create additional child handles on additional calls to **Supported()** and **Start()**.

If *ControllerHandle* is not supported by *This*, then **EFI_UNSUPPORTED** is returned.

If *This* is a bus driver that creates child handles with an **EFI_DEVICE_PATH_PROTOCOL**, then *ControllerHandle* must support the **EFI_DEVICE_PATH**. If it does not, then **EFI_UNSUPPORTED** is returned.

If *ControllerHandle* is supported by *This*, and *This* is a device driver, then **EFI_SUCCESS** is returned.

If *ControllerHandle* is supported by *This*, and *This* is a bus driver, and *RemainingDevicePath* is **NULL**, then **EFI_SUCCESS** is returned.

If *ControllerHandle* is supported by *This*, and *This* is a bus driver, and *RemainingDevicePath* is not **NULL**, then *RemainingDevicePath* must be analyzed. If *RemainingDevicePath* starts with an EFI Device Path node that the bus driver recognizes and supports, then **EFI_SUCCESS** is returned. Otherwise, **EFI_UNSUPPORTED** is returned.

The **Supported()** function is designed to be invoked from the EFI boot service **ConnectController()**. As a result, much of the error checking on the parameters to **Supported()** has been moved into this common boot service. It is legal to call **Supported()** from other locations, but the following calling restrictions must be followed or the system behavior will not be deterministic.

ControllerHandle must be a valid **EFI_HANDLE**. If *RemainingDevicePath* is not **NULL**, then it must be a pointer to a naturally aligned **EFI_DEVICE_PATH** that contains at least one device path node other than the end node.

Status Codes Returned

EFI_SUCCESS	The device specified by *ControllerHandle* and *RemainingDevicePath* is supported by the driver specified by *This*.
EFI_ALREADY_STARTED	The device specified by *ControllerHandle* and *RemainingDevicePath* is already being managed by the driver specified by *This*.
EFI_ACCESS_DENIED	The device specified by *ControllerHandle* and *RemainingDevicePath* is already being managed by a different driver or an application that requires exclusive access.
EFI_UNSUPPORTED	The device specified by *ControllerHandle* and *RemainingDevicePath* is not supported by the driver specified by *This*.

Examples

```
extern EFI_GUID              gEfiDriverBindingProtocolGuid;
EFI_HANDLE                   DriverImageHandle;
EFI_HANDLE                   ControllerHandle;
EFI_DRIVER_BINDING_PROTOCOL  *DriverBinding;
EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath;

//
// Use the DriverImageHandle to get the Driver Binding Protocol instance
//
Status = gBS->OpenProtocol (
                  DriverImageHandle,
                  &gEfiDriverBindingProtocolGuid,
                  &DriverBinding,
                  DriverImageHandle,
                  NULL,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// EXAMPLE #1
//
// Use the Driver Binding Protocol instance to test to see if the
// driver specified by DriverImageHandle supports the controller
// specified by ControllerHandle
//
Status = DriverBinding->Supported (
                          DriverBinding,
                          ControllerHandle,
                          NULL
                          );
return Status;
```

```
//
// EXAMPLE #2
//
// The RemainingDevicePath parameter can be used to initialize only
// the minimum devices required to boot.  For example, maybe we only
// want to initialize 1 hard disk on a SCSI channel.  If DriverImageHandle
// is a SCSI Bus Driver, and ControllerHandle is a SCSI Controller, and
// we only want to create a child handle for PUN=3 and LUN=0, then the
// RemainingDevicePath would be SCSI(3,0)/END.  The following example
// would return EFI_SUCCESS if the SCSI driver supports creating the
// child handle for PUN=3, LUN=0.  Otherwise it would return an error.
//
Status = DriverBinding->Supported (
                          DriverBinding,
                          ControllerHandle,
                          RemainingDevicePath
                          );
return Status;
```

Pseudo Code

Listed below are the algorithms for the **Supported()** function for three different types of drivers. How the **Start()** function of a driver is implemented can affect how the **Supported()** function is implemented. All of the services in the **EFI_DRIVER_BINDING_PROTOCOL** need to work together to make sure that all resources opened or allocated in **Supported()** and **Start()** are released in **Stop()**.

The first algorithm is a simple device driver that does not create any additional handles. It only attaches one or more protocols to an existing handle. The second is a bus driver that always creates all of its child handles on the first call to **Start()**. The third is a more advanced bus driver that can either create one child handles at a time on successive calls to **Start()**, or it can create all of its child handles or all of the remaining child handles in a single call to **Start()**.

Device Driver:

1. Ignore the parameter *RemainingDevicePath*.
2. Open all required protocols with **OpenProtocol()**. A standard driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER**. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE**.
3. If any of the calls to **OpenProtocol()** in (2) returned an error, then close all of the protocols opened in (2) with **CloseProtocol()**, and return the status code from the call to **OpenProtocol()** that returned an error.
4. Use the protocol instances opened in (2) to test to see if this driver supports the controller. Sometimes, just the presence of the protocols is enough of a test. Other times, the services of the protocols opened in (2) are used to further check the identity of the controller. If any of these tests fails, then close all the protocols opened in (2) with **CloseProtocol()** and return **EFI_UNSUPPORTED**.
5. Close all protocols opened in (2) with **CloseProtocol()**.
6. Return **EFI_SUCCESS**.

Bus Driver that creates all of its child handles on the first call to Start():

1. Check the contents of the first Device Path Node of *RemainingDevicePath* to make sure it is a legal Device Path Node for this bus driver's children. If it is not, then return **EFI_UNSUPPORTED**.
2. Open all required protocols with **OpenProtocol()**. A standard driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER**. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE**.
3. If any of the calls to **OpenProtocol()** in (2) returned an error, then close all of the protocols opened in (2) with **CloseProtocol()**, and return the status code from the call to **OpenProtocol()** that returned an error.
4. Use the protocol instances opened in (2) to test to see if this driver supports the controller. Sometimes, just the presence of the protocols is enough of a test. Other times, the services of the protocols opened in (2) are used to further check the identity of the controller. If any of these tests fails, then close all the protocols opened in (2) with **CloseProtocol()** and return **EFI_UNSUPPORTED**.
5. Close all protocols opened in (2) with **CloseProtocol()**.
6. Return **EFI_SUCCESS**.

Bus Driver that is able to create all or one of its child handles on each call to Start():

1. Check the contents of the first Device Path Node of *RemainingDevicePath* to make sure it is a legal Device Path Node for this bus driver's children. If it is not, then return **EFI_UNSUPPORTED**.
2. Open all required protocols with **OpenProtocol()**. A standard driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER**. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE**.
3. If any of the calls to **OpenProtocol()** in (2) failed with an error other than **EFI_ALREADY_STARTED**, then close all of the protocols opened in (2) that did not return **EFI_ALREADY_STARTED** with **CloseProtocol()**, and return the status code from the **OpenProtocol()** call that returned an error.
4. Use the protocol instances opened in (2) to test to see if this driver supports the controller. Sometimes, just the presence of the protocols is enough of a test. Other times, the services of the protocols opened in (2) are used to further check the identity of the controller. If any of these tests fails, then close all the protocols opened in (2) that did not return **EFI_ALREADY_STARTED** with **CloseProtocol()** and return **EFI_UNSUPPORTED**.
5. Close all protocols opened in (2) that did not return **EFI_ALREADY_STARTED** with **CloseProtocol()**.
6. Return **EFI_SUCCESS**.

Listed below is sample code of the **Supported()** function of device driver for a device on the XYZ bus. The XYZ bus is abstracted with the **EFI_XYZ_IO_PROTOCOL**. Just the presence of the **EFI_XYZ_IO_PROTOCOL** on *ControllerHandle* is enough to determine if this driver supports *ControllerHandle*. The **gBS** variable is initialized in this driver's entry point. See Chapter 4.

```
extern EFI_GUID          gEfiXyzIoProtocol;
EFI_BOOT_SERVICES_TABLE  *gBS;

EFI_STATUS
AbcSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )

{
  EFI_STATUS           Status;
  EFI_XYZ_IO_PROTOCOL  *XyzIo;

  Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiXyzIoProtocol,
                  &XyzIo,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_BY_DRIVER
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  gBS->CloseProtocol (
         ControllerHandle,
         &gEfiXyzIoProtocol,
         This->DriverBindingHandle,
         ControllerHandle
         );

  return EFI_SUCCESS;
}
```

EFI_DRIVER_BINDING_PROTOCOL.Start()

Summary

Starts a device controller or a bus controller. The **Start()** and **Stop()** services of the **EFI_DRIVER_BINDING_PROTOCOL** mirror each other.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_START) (
  IN  EFI_DRIVER_BINDING_PROTOCOL   *This,
  IN  EFI_HANDLE                    ControllerHandle,
  IN  EFI_DEVICE_PATH               *RemainingDevicePath  OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_DRIVER_BINDING_PROTOCOL** instance.
ControllerHandle	The handle of the controller to start. This handle must support a protocol interface that supplies an I/O abstraction to the driver.
RemainingDevicePath	A pointer to the remaining portion of a device path. This parameter is ignored by device drivers, and is optional for bus drivers. For a bus driver, if this parameter is **NULL**, then handles for all the children of *Controller* are created by this driver. If this parameter is not **NULL**, then only the handle for the child device specified by the first Device Path Node of *RemainingDevicePath* is created by this driver.

Description

This function starts the device specified by *Controller* with the driver specified by *This*. Whatever resources are allocated in **Start()** must be freed in **Stop()**. For example, every **AllocatePool()**, **AllocatePages()**, **OpenProtocol()**, and **InstallProtocolInterface()** in **Start()** must be matched with a **FreePool()**, **FreePages()**, **CloseProtocol()**, and **UninstallProtocolInterface()** in **Stop()**.

If *Controller* is started, then **EFI_SUCCESS** is returned. If *Controller* cannot be started due to a device error, then **EFI_DEVICE_ERROR** is returned. If there are not enough resources to start the device or bus specified by *Controller*, then **EFI_OUT_OF_RESOURCES** is returned.

If the driver specified by *This* is a device driver, then *RemainingDevicePath* is ignored.

If the driver specified by *This* is a bus driver, and *RemainingDevicePath* is **NULL**, then all of the children of *Controller* are discovered and enumerated, and a device handle is created for each child.

If the driver specified by *This* is a bus driver that is capable of creating one child handle at a time and *RemainingDevicePath* is not **NULL**, then only the device handle for the child device specified by *RemainingDevicePath* is created. Depending on the bus type, all of the child devices may need to be discovered and enumerated, but only device handle for the one child specified by *RemainingDevicePath* shall be created.

The **Start()** function is designed to be invoked from the EFI boot service **ConnectController()**. As a result, much of the error checking on the parameters to **Start()** has been moved into this common boot service. It is legal to call **Start()** from other locations, but the following calling restrictions must be followed or the system behavior will not be deterministic.

1. *ControllerHandle* must be a valid **EFI_HANDLE**.
2. If *RemainingDevicePath* is not **NULL**, then it must be a pointer to a naturally aligned **EFI_DEVICE_PATH** that contains at least one device path node other than the end node.
3. Prior to calling **Start()**, the **Supported()** function for the driver specified by *This* must have been called with the same calling parameters, and **Supported()** must have returned **EFI_SUCCESS**.

Status Codes Returned

EFI_SUCCESS	The device was started.
EFI_DEVICE_ERROR	The device could not be started due to a device error.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

Examples

```
extern EFI_GUID                  gEfiDriverBindingProtocolGuid;
EFI_HANDLE                       DriverImageHandle;
EFI_HANDLE                       ControllerHandle;
EFI_DRIVER_BINDING_PROTOCOL      *DriverBinding;
EFI_DEVICE_PATH_PROTOCOL         *RemainingDevicePath;

//
// Use the DriverImageHandle to get the Driver Binding Protocol instance
//
Status = gBS->OpenProtocol (
                  DriverImageHandle,
                  &gEfiDriverBindingProtocolGuid,
                  &DriverBinding,
                  DriverImageHandle,
                  NULL,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
if (EFI_ERROR (Status)) {
  return Status;
}
```

```
//
// EXAMPLE #1
//
// Use the Driver Binding Protocol instance to test to see if the
// driver specified by DriverImageHandle supports the controller
// specified by ControllerHandle
//
Status = DriverBinding->Supported (
                          DriverBinding,
                          ControllerHandle,
                          NULL
                          );
if (!EFI_ERROR (Status)) {
  Status = DriverBinding->Start (
                            DriverBinding,
                            ControllerHandle,
                            NULL
                            );
}

return Status;

//
// EXAMPLE #2
//
// The RemainingDevicePath parameter can be used to initialize only
// the minimum devices required to boot.  For example, maybe we only
// want to initialize 1 hard disk on a SCSI channel.  If DriverImageHandle
// is a SCSI Bus Driver, and ControllerHandle is a SCSI Controller, and
// we only want to create a child handle for PUN=3 and LUN=0, then the
// RemainingDevicePath would be SCSI(3,0)/END.  The following example
// would return EFI_SUCCESS if the SCSI driver supports creating the
// child handle for PUN=3, LUN=0.  Otherwise it would return an error.
//
Status = DriverBinding->Supported (
                          DriverBinding,
                          ControllerHandle,
                          RemainingDevicePath
                          );
if (!EFI_ERROR (Status)) {
  Status = DriverBinding->Start (
                            DriverBinding,
                            ControllerHandle,
                            RemainingDevicePath
                            );
}

return Status;
```

Pseudo Code

Listed below are the algorithms for the `Start()` function for three different types of drivers. How the `Start()` function of a driver is implemented can affect how the `Supported()` function is implemented. All of the services in the `EFI_DRIVER_BINDING_PROTOCOL` need to work together to make sure that all resources opened or allocated in `Supported()` and `Start()` are released in `Stop()`.

The first algorithm is a simple device driver that does not create any additional handles. It only attaches one or more protocols to an existing handle. The second is a simple bus driver that always creates all of its child handles on the first call to `Start()`. It does not attach any additional protocols to the handle for the bus controller. The third is a more advanced bus driver that can either create one child handles at a time on successive calls to `Start()`, or it can create all of its child handles or all of the remaining child handles in a single call to `Start()`. Once again, it does not attach any additional protocols to the handle for the bus controller.

Device Driver:

1. Ignore the parameter *RemainingDevicePath*.
2. Open all required protocols with `OpenProtocol()`. A standard driver should use an *Attribute* of `EFI_OPEN_PROTOCOL_BY_DRIVER`. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an *Attribute* of `EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE`. It must use the same *Attribute* value that was used in `Supported()`.
3. If any of the calls to `OpenProtocol()` in (2) returned an error, then close all of the protocols opened in (2) with `CloseProtocol()`, and return the status code from the call to `OpenProtocol()` that returned an error.
4. Initialize the device specified by *ControllerHandle*. If an error occurs, close all of the protocols opened in (2) with `CloseProtocol()`, and return `EFI_DEVICE_ERROR`.
5. Allocate and initialize all of the data structures that this driver requires to manage the device specified by *ControllerHandle*. This would include space for public protocols and space for any additional private data structures that are related to *ControllerHandle*. If an error occurs allocating the resources, then close all of the protocols opened in (2) with `CloseProtocol()`, and return `EFI_OUT_OF_RESOURCES`.
6. Install all the new protocol interfaces onto *ControllerHandle* using `InstallMultipleProtocolInterfaces()`. If an error occurs, close all of the protocols opened in (1) with `CloseProtocol()`, and return the error from `InstallMultipleProtocolInterfaces`().
7. Return `EFI_SUCCESS`.

Bus Driver that creates all of its child handles on the first call to Start():

1. Ignore the parameter *RemainingDevicePath*.
2. Open all required protocols with **OpenProtocol()**. A standard driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER**. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE**. It must use the same *Attribute* value that was used in **Supported()**.
3. If any of the calls to **OpenProtocol()** in (2) returned an error, then close all of the protocols opened in (2) with **CloseProtocol()**, and return the status code from the call to **OpenProtocol()** that returned an error.
4. Initialize the device specified by *ControllerHandle*. If an error occurs, close all of the protocols opened in (2) with **CloseProtocol()**, and return **EFI_DEVICE_ERROR**.
5. Discover all the child devices of the bus controller specified by *ControllerHandle*.
6. If the bus requires it, allocate resources to all the child devices of the bus controller specified by *ControllerHandle*.
7. FOR each child C of *ControllerHandle:*
    a. Allocate and initialize all of the data structures that this driver requires to manage the child device C. This would include space for public protocols and space for any additional private data structures that are related to the child device C. If an error occurs allocating the resources, then close all of the protocols opened in (2) with **CloseProtocol()**, and return **EFI_OUT_OF_RESOURCES**.
    b. If the bus driver creates device paths for the child devices, then create a device path for the child C based upon the device path attached to *ControllerHandle*.
    c. Initialize the child device C. If an error occurs, close all of the protocols opened in (2) with **CloseProtocol()**, and return **EFI_DEVICE_ERROR**.
    d. Create a new handle for C, and install the protocol interfaces for child device C using **InstallMultipleProtocolInterfaces()**. This may include the **EFI_DEVICE_PATH** protocol.
    e. Call **OpenProtocol()** on behalf of the child C with an *Attribute* of **EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER**.
8. END FOR
9. Return **EFI_SUCCESS**.

Bus Driver that is able to create all or one of its child handles on each call to Start():

1. Open all required protocols with **OpenProtocol()**. A standard driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER**. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an *Attribute* of **EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE**. It must use the same *Attribute* value that was used in **Supported()**.
2. If any of the calls to **OpenProtocol()** in (1) returned an error, then close all of the protocols opened in (1) with **CloseProtocol()**, and return the status code from the call to **OpenProtocol()** that returned an error.
3. Initialize the device specified by *ControllerHandle*. If an error occurs, close all of the protocols opened in (1) with **CloseProtocol()**, and return **EFI_DEVICE_ERROR**.
4. IF *RemainingDevicePath* is not **NULL**, THEN
   a. C is the child device specified by *RemainingDevicePath*.
   b. Allocate and initialize all of the data structures that this driver requires to manage the child device C. This would include space for public protocols and space for any additional private data structures that are related to the child device C. If an error occurs allocating the resources, then close all of the protocols opened in (1) with **CloseProtocol()**, and return **EFI_OUT_OF_RESOURCES**.
   c. If the bus driver creates device paths for the child devices, then create a device path for the child C based upon the device path attached to *ControllerHandle*.
   d. Initialize the child device C.
   e. Create a new handle for C, and install the protocol interfaces for child device C using **InstallMultipleProtocolInterfaces()**. This may include the **EFI_DEVICE_PATH** protocol.
   f. Call **OpenProtocol()** on behalf of the child C with an *Attribute* of **EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER**.

ELSE

5. Discover all the child devices of the bus controller specified by *ControllerHandle*.
6. If the bus requires it, allocate resources to all the child devices of the bus controller specified by *ControllerHandle*.
7. FOR each child C of *ControllerHandle*
   a. Allocate and initialize all of the data structures that this driver requires to manage the child device C. This would include space for public protocols and space for any additional private data structures that are related to the child device C. If an error occurs allocating the resources, then close all of the protocols opened in (1) with **CloseProtocol()**, and return **EFI_OUT_OF_RESOURCES**.
   b. If the bus driver creates device paths for the child devices, then create a device path for the child C based upon the device path attached to *ControllerHandle*.
   c. Initialize the child device C.
   d. Create a new handle for C, and install the protocol interfaces for child device C using **InstallMultipleProtocolInterfaces()**. This may include the **EFI_DEVICE_PATH** protocol.

    e. Call **OpenProtocol()** on behalf of the child C with an *Attribute* of **EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER**.
  8. END FOR
9. END IF
10. Return **EFI_SUCCESS**.

Listed below is sample code of the **Start()** function of a device driver for a device on the XYZ bus. The XYZ bus is abstracted with the **EFI_XYZ_IO_PROTOCOL**. This driver does allow the **EFI_XYZ_IO_PROTOCOL** to be shared with other drivers, and just the presence of the **EFI_XYZ_IO_PROTOCOL** on *ControllerHandle* is enough to determine if this driver supports *ControllerHandle*. This driver installs the **EFI_ABC_IO_PROTOCOL** on *ControllerHandle*. The **gBS** variable is initialized in this driver's entry point as shown in the EFI Driver Model examples in Chapter 4.

```
extern EFI_GUID          gEfiXyzIoProtocol;
extern EFI_GUID          gEfiAbcIoProtocol;
EFI_BOOT_SERVICES_TABLE  *gBS;

EFI_STATUS
AbcStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
)

{
  EFI_STATUS           Status;
  EFI_XYZ_IO_PROTOCOL  *XyzIo;
  EFI_ABC_DEVICE       AbcDevice;

  //
  // Open the Xyz I/O Protocol that this driver consumes
  //
  Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfiXyzIoProtocol,
                  &XyzIo,
                  This->DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_BY_DRIVER
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Allocate and zero a private data structure for the Abc device.
  //
  Status = gBS->AllocatePool (
                  EfiBootServicesData,
                  sizeof (EFI_ABC_DEVICE),
                  &AbcDevice
                  );
  if (EFI_ERROR (Status)) {
    goto ErrorExit;
  }
  ZeroMem (AbcDevice, sizeof (EFI_ABC_DEVICE));

  //
  // Initialize the contents of the private data structure for the Abc device.
  // This includes the XyzIo protocol instance and other private data fields
  // and the EFI_ABC_IO_PROTOCOL instance that will be installed.
  //
  AbcDevice->Signature       = EFI_ABC_DEVICE_SIGNATURE;
  AbcDevice->XyzIo           = XyzIo;

  AbcDevice->PrivateData1    = PrivateValue1;
  AbcDevice->PrivateData2    = PrivateValue2;
  . . .
  AbcDevice->PrivateDataN    = PrivateValueN;

  AbcDevice->AbcIo.Revision  = EFI_ABC_IO_PROTOCOL_REVISION;
  AbcDevice->AbcIo.Func1     = AbcIoFunc1;
  AbcDevice->AbcIo.Func2     = AbcIoFunc2;
  . . .
  AbcDevice->AbcIo.FuncN     = AbcIoFuncN;

  AbcDevice->AbcIo.Data1     = Value1;
  AbcDevice->AbcIo.Data2     = Value2;
  . . .
  AbcDevice->AbcIo.DataN     = ValueN;

  //
  // Install protocol interfaces for the ABC I/O device.
  //
  Status = gBS->InstallMultipleProtocolInterfaces (
                  &ControllerHandle,
                  &gEfiAbcIoProtocolGuid, &AbcDevice->AbcIo,
                  NULL
                  );
  if (EFI_ERROR (Status)) {
    goto ErrorExit;
  }

  return EFI_SUCCESS;

ErrorExit:
  //
  // When there is an error, the private data structures need to be freed and
  // the protocols that were opened need to be closed.
  //
  if (AbcDevice != NULL) {
    gBS->FreePool (AbcDevice);
  }
  gBS->CloseProtocol (
         ControllerHandle,
         &gEfiXyzIoProtocolGuid,
         This->DriverBindingHandle,
         ControllerHandle
         );
  return Status;
}
```

EFI_DRIVER_BINDING_PROTOCOL.Stop()

Summary

Stops a device controller or a bus controller. The **Start()** and **Stop()** services of the **EFI_DRIVER_BINDING_PROTOCOL** mirror each other.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_STOP) (
  IN  EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN  EFI_HANDLE                   ControllerHandle,
  IN  UINTN                        NumberOfChildren,
  IN  EFI_HANDLE                   *ChildHandleBuffer  OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_DRIVER_BINDING_PROTOCOL** instance. Type **EFI_DRIVER_BINDING_PROTOCOL** is defined in section 9.1.
ControllerHandle	A handle to the device being stopped. The handle must support a bus specific I/O protocol for the driver to use to stop the device.
NumberOfChildren	The number of child device handles in *ChildHandleBuffer*.
ChildHandleBuffer	An array of child handles to be freed. May be **NULL** if *NumberOfChildren* is 0.

Description

This function performs different operations depending on the parameter *NumberOfChildren*. If *NumberOfChildren* is not zero, then the driver specified by *This* is a bus driver, and it is being requested to free one or more of its child handles specified by *NumberOfChildren* and *ChildHandleBuffer*. If all of the child handles are freed, then **EFI_SUCCESS** is returned. If *NumberOfChildren* is zero, then the driver specified by *This* is either a device driver or a bus driver, and it is being requested to stop the controller specified by *ControllerHandle*. If *ControllerHandle* is stopped, then **EFI_SUCCESS** is returned. In either case, this function is required to undo what was performed in **Start()**. Whatever resources are allocated in **Start()** must be freed in **Stop()**. For example, every **AllocatePool()**, **AllocatePages()**, **OpenProtocol()**, and **InstallProtocolInterface()** in **Start()** must be matched with a **FreePool()**, **FreePages()**, **CloseProtocol()**, and **UninstallProtocolInterface()** in **Stop()**.

If *ControllerHandle* cannot be stopped, then **EFI_DEVICE_ERROR** is returned. If, for some reason, there are not enough resources to stop *ControllerHandle*, then **EFI_OUT_OF_RESOURCES** is returned. If *ControllerHandle* was not started by the driver specified by *This*, then **EFI_UNSUPPORTED** is returned.

The **Stop()** function is designed to be invoked from the EFI boot service **DisconnectController()**. As a result, much of the error checking on the parameters to **Stop()** has been moved into this common boot service. It is legal to call **Stop()** from other locations, but the following calling restrictions must be followed or the system behavior will not be deterministic.

1. *ControllerHandle* must be a valid **EFI_HANDLE** that was used on a previous call to this same driver's **Start()** function.
2. The first *NumberOfChildren* handles of *ChildHandleBuffer* must all be a valid **EFI_HANDLE**. In addition, all of these handles must have been created in this driver's **Start()** function, and the **Start()** function must have called **OpenProtocol()** on *ControllerHandle* with an *Attribute* of **EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER**.

Status Codes Returned

EFI_SUCCESS	The device was stopped.
EFI_DEVICE_ERROR	The device could not be stopped due to a device error.

Examples

```
extern EFI_GUID                gEfiDriverBindingProtocolGuid;
EFI_HANDLE                     DriverImageHandle;
EFI_HANDLE                     ControllerHandle;
EFI_HANDLE                     ChildHandle;
EFI_DRIVER_BINDING_PROTOCOL    *DriverBinding;

//
// Use the DriverImageHandle to get the Driver Binding Protocol instance
//
Status = gBS->OpenProtocol (
                  DriverImageHandle,
                  &gEfiDriverBindingProtocolGuid,
                  &DriverBinding,
                  DriverImageHandle,
                  NULL,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use the Driver Binding Protocol instance to free the child
// specified by ChildHandle.  Then, use the Driver Binding
// Protocol to stop ControllerHandle.
//
```

```
Status = DriverBinding->Stop (
                          DriverBinding,
                          ControllerHandle,
                          1,
                          &ChildHandle
                          );

Status = DriverBinding->Stop (
                          DriverBinding,
                          ControllerHandle,
                          0,
                          NULL
                          );
```

Pseudo Code

Device Driver:

```
1. Uninstall all the protocols that were installed onto ControllerHandle in Start().
2. Close all the protocols that were opened on behalf of ControllerHandle in Start().
3. Free all the structures that were allocated on behalf of ControllerHandle in Start().
4. Return EFI_SUCCESS.
```

Bus Driver that creates all of its child handles on the first call to Start():

Bus Driver that is able to create all or one of its child handles on each call to Start():

```
1. IF NumberOfChildren is zero THEN:
   a. Uninstall all the protocols that were installed onto ControllerHandle in Start().
   b. Close all the protocols that were opened on behalf of ControllerHandle in Start().
   c. Free all the structures that were allocated on behalf of ControllerHandle in
      Start().
2. ELSE
   a. FOR each child C in ChildHandleBuffer
   —  Uninstall all the protocols that were installed onto C in Start().
   —  Close all the protocols that were opened on behalf of C in Start().
   —  Free all the structures that were allocated on behalf of C in Start().
   b. END FOR
3. END IF
4. Return EFI_SUCCESS.
```

Listed below is sample code of the **`Stop()`** function of a device driver for a device on the XYZ bus. The XYZ bus is abstracted with the **`EFI_XYZ_IO_PROTOCOL`**. This driver does allow the **`EFI_XYZ_IO_PROTOCOL`** to be shared with other drivers, and just the presence of the **`EFI_XYZ_IO_PROTOCOL`** on *`ControllerHandle`* is enough to determine if this driver supports *`ControllerHandle`*. This driver installs the **`EFI_ABC_IO_PROTOCOL`** on *`ControllerHandle`* in **`Start()`**. The **`gBS`** variable is initialized in this driver's entry point. See Chapter 4.

```
extern EFI_GUID          gEfiXyzIoProtocol;
extern EFI_GUID          gEfiAbcIoProtocol;
EFI_BOOT_SERVICES_TABLE  *gBS;

EFI_STATUS
AbcStop (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle
  IN UINTN                        NumberOfChildren,
  IN EFI_HANDLE                   *ChildHandleBuffer  OPTIONAL
  )

{
  EFI_STATUS             Status;
  EFI_ABC_IO             AbcIo;
  EFI_ABC_DEVICE         AbcDevice;

  //
  // Get our context back
  //
  Status = gBS->OpenProtocol (
                 ControllerHandle,
                 &gEfiAbcIoProtocolGuid,
                 &AbcIo,
                 This->DriverBindingHandle,
                 ControllerHandle,
                 EFI_OPEN_PROTOCOL_GET_PROTOCOL
                 );
  if (EFI_ERROR (Status)) {
    return EFI_UNSUPPORTED;
  }

  //
  // Use Containment Record Macro to get AbcDevice structure from
  // a pointer to the AbcIo structure within the AbcDevice structure.
  //
  AbcDevice = ABC_IO_PRIVATE_DATA_FROM_THIS (AbcIo);
  //
  // Uninstall the protocol installed in Start()
  //
  Status = gBS->UninstallMultipleProtocolInterfaces (
                  ControllerHandle,
                  &gEfiAbcIoProtocolGuid, &AbcDevice->AbcIo,
                  NULL
                  );
  if (!EFI_ERROR (Status)) {

    //
    // Close the protocol opened in Start()
    //
    Status = gBS->CloseProtocol (
                    ControllerHandle,
                    &gEfiXyzIoProtocolGuid,
                    This->DriverBindingHandle,
                    ControllerHandle
                    );

    //
    // Free the structure allocated in Start().
    //
    gBS->FreePool (AbcDevice);
  }

  return Status;

}
```

9.2 EFI Platform Driver Override Protocol

This section provides a detailed description of the **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL**. This protocol can override the default algorithm for matching drivers to controllers.

EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL

Summary

This protocol matches one or more drivers to a controller. A platform driver produces this protocol, and it is installed on a separate handle. This protocol is used by the **ConnectController()** boot service to select the best driver for a controller. All of the drivers returned by this protocol have a higher precedence than drivers found from an EFI Bus Specific Driver Override Protocol or drivers found from the general EFI Driver Binding search algorithm. If more than one driver is returned by this protocol, then the drivers are returned in order from highest precedence to lowest precedence.

GUID

```
#define EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL_GUID \
  { 0x6b30c738,0xa391,0x11d4,0x9a,0x3b,0x00,0x90,0x27,0x3f,0xc1,0x4d }
```

Protocol Interface Structure

```
typedef struct _EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL {
  EFI_PLATFORM_DRIVER_OVERRIDE_GET_DRIVER        GetDriver;
  EFI_PLATFORM_DRIVER_OVERRIDE_GET_DRIVER_PATH   GetDriverPath;
  EFI_PLATFORM_DRIVER_OVERRIDE_DRIVER_LOADED     DriverLoaded;
} EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL;
```

Parameters

GetDriver	Retrieves the image handle of a platform override driver for a controller in the system. See the **GetDriver()** function description.
GetDriverPath	Retrieves the device path of a platform override driver for a controller in the system. See the **GetDriverPath()** function description.
DriverLoaded	This function is used after a driver has been loaded using a device path returned by **GetDriverPath()**. This function associates a device path to an image handle, so the image handle can be returned the next time that **GetDriver()** is called for the same controller. See the **DriverLoaded()** function description.

Description

The **`EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL`** is used by the EFI boot service **`ConnectController()`** to determine if there is a platform specific driver override for a controller that is about to be started. The bus drivers in a platform will use a bus defined matching algorithm for matching drivers to controllers. This protocol allows the platform to override the bus driver's default driver matching algorithm. This protocol can be used to specify the drivers for on-board devices whose drivers may be in a system ROM not directly associated with the on-board controller, or it can even be used to manage the matching of drivers and controllers in add-in cards. This can be very useful if there are two adapters that are identical except for the revision of the driver in the adapter's ROM. This protocol, along with a platform configuration utility, could specify which of the two drivers to use for each of the adapters.

The drivers that this protocol returns can be either in the form of an image handle or a device path. **`ConnectController()`** can only use image handles, so **`ConnectController()`** is required to use the **`GetDriver()`** service. A different component, such as the Boot Manager, will have to use the **`GetDriverPath()`** service to retrieve the list of drivers that need to be loaded from I/O devices. Once a driver has been loaded and started, this same component can use the **`DriverLoaded()`** service to associate the device path of a driver with the image handle of the loaded driver. Once this association has been established, the image handle can then be returned by the **`GetDriver()`** service the next time it is called by **`ConnectController()`**.

EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL.GetDriver()

Summary

Retrieves the image handle of the platform override driver for a controller in the system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PLATFORM_DRIVER_OVERRIDE_GET_DRIVER) (
  IN     EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL  *This,
  IN     EFI_HANDLE                             ControllerHandle,
  IN OUT EFI_HANDLE                             *DriverImageHandle
  );
```

Parameters

This	A pointer to the **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL** instance.
ControllerHandle	The device handle of the controller to check if a driver override exists.
DriverImageHandle	On input, a pointer to the previous driver image handle returned by **GetDriver()**. On output, a pointer to the next driver image handle. Passing in a **NULL**, will return the first driver image handle for *ControllerHandle*.

Description

This function is used to retrieve a driver image handle that is selected in a platform specific manner. The first driver image handle is retrieved by passing in a *DriverImageHandle* value of **NULL**. This will cause the first driver image handle to be returned in *DriverImageHandle*. On each successive call, the previous value of *DriverImageHandle* must be passed in. If a call to this function returns a valid driver image handle, then **EFI_SUCCESS** is returned. This process is repeated until **EFI_NOT_FOUND** is returned. If a *DriverImageHandle* is passed in that was not returned on a prior call to this function, then **EFI_INVALID_PARAMETER** is returned. If *ControllerHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. The first driver image handle has the highest precedence, and the last driver image handle has the lowest precedence. This ordered list of driver image handles is used by the boot service **ConnectController()** to search for the best driver for a controller.

Status Codes Returned

EFI_SUCCESS	The driver override for *ControllerHandle* was returned in *DriverImageHandle*.
EFI_NOT_FOUND	A driver override for *ControllerHandle* was not found.
EFI_INVALID_PARAMETER	The handle specified by *ControllerHandle* is not a valid handle.
EFI_INVALID_PARAMETER	*DriverImageHandle* is not a handle that was returned on a previous call to **GetDriver()**.

EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL.GetDriverPath()

Summary

Retrieves the device path of the platform override driver for a controller in the system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PLATFORM_DRIVER_OVERRIDE_GET_DRIVER_PATH) (
  IN     EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL   *This,
  IN     EFI_HANDLE                             ControllerHandle,
  IN OUT EFI_DEVICE_PATH_PROTOCOL               **DriverImagePath
  );
```

Parameters

This	A pointer to the **EFI_PLATFORM_DRIVER_OVERRIDE_ PROTOCOL** instance.
ControllerHandle	The device handle of the controller to check if a driver override exists.
DriverImagePath	On input, a pointer to the previous driver device path returned by **GetDriverPath()**. On output, a pointer to the next driver device path. Passing in a pointer to **NULL**, will return the first driver device path for *ControllerHandle*.

Description

This function is used to retrieve a driver device path that is selected in a platform specific manner. The first driver device path is retrieved by passing in a *DriverImagePath* value that is a pointer to **NULL**. This will cause the first driver device path to be returned in *DriverImagePath*. On each successive call, the previous value of *DriverImagePath* must be passed in. If a call to this function returns a valid driver device path, then **EFI_SUCCESS** is returned. This process is repeated until **EFI_NOT_FOUND** is returned. If a *DriverImagePath* is passed in that was not returned on a prior call to this function, then **EFI_INVALID_PARAMETER** is returned. If *ControllerHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. The first driver device path has the highest precedence, and the last driver device path has the lowest precedence. This ordered list of driver device paths is used by a platform specific component, such as the EFI Boot Manager, to load and start the platform override drivers by using the EFI boot services **LoadImage()** and **StartImage()**. Each time one of these drivers is loaded and started, the **DriverLoaded()** service is called.

Status Codes Returned

EFI_SUCCESS	The driver override for *ControllerHandle* was returned in *DriverImagePath*.
EFI_NOT_FOUND	A driver override for *ControllerHandle* was not found.
EFI_INVALID_PARAMETER	The handle specified by *ControllerHandle* is not a valid handle.
EFI_INVALID_PARAMETER	*DriverImagePath* is not a device path that was returned on a previous call to **GetDriverPath()**.

EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL.DriverLoaded()

Summary

Used to associate a driver image handle with a device path that was returned on a prior call to the **GetDriverPath()** service. This driver image handle will then be available through the **GetDriver()** service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PLATFORM_DRIVER_OVERRIDE_DRIVER_LOADED) (
  IN EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL  *This,
  IN EFI_HANDLE                            ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL              *DriverImagePath,
  IN EFI_HANDLE                            DriverImageHandle
  );
```

Parameters

This	A pointer to the **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL** instance.
ControllerHandle	The device handle of a controller. This must match the controller handle that was used in a prior call to **GetDriver()** to retrieve *DriverImagePath*.
DriverImagePath	A pointer to the driver device path that was returned in a prior call to **GetDriverPath()**.
DriverImageHandle	The driver image handle that was returned by **LoadImage()** when the driver specified by *DriverImagePath* was loaded into memory.

Description

This function associates the image handle specified by *DriverImageHandle* with the device path of a driver specified by *DriverImagePath*. *DriverImagePath* must be a value that was returned on a prior call to **GetDriverPath()** for the controller specified by *ControllerHandle*. Once this association has been established, then the service **GetDriver()** must return *DriverImageHandle* as one of the override drivers for the controller specified by *ControllerHandle*.

If the association between the image handle specified by *DriverImageHandle* and the device path specified by *DriverImagePath* is established for the controller specified by *ControllerHandle*, then **EFI_SUCCESS** is returned. If *ControllerHandle* is not a valid **EFI_HANDLE**, or *DriverImagePath* is not a valid device path, or *DriverImageHandle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. If *DriverImagePath* is not a device path that was returned on a prior call to **GetDriver()** for the controller specified by *ControllerHandle*, then **EFI_NOT_FOUND** is returned.

Status Codes Returned

EFI_SUCCESS	The association between *DriverImagePath* and *DriverImageHandle* was established for the controller specified by *ControllerHandle*.
EFI_NOT_FOUND	*DriverImagePath* is not a device path that was returned on a prior call to **GetDriverPath()** for the controller specified by *ControllerHandle*.
EFI_INVALID_PARAMETER	*ControllerHandle* is not a valid device handle.
EFI_INVALID_PARAMETER	*DriverImagePath* is not a valid device path.
EFI_INVALID_PARAMETER	*DriverImageHandle* is not a valid image handle.

9.3 EFI Bus Specific Driver Override Protocol

This section provides a detailed description of the **EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL**. Bus drivers that have a bus specific algorithm for matching drivers to controllers are required to produce this protocol for each controller. For example, a PCI Bus Driver will produce an instance of this protocol for every PCI controller that has a PCI option ROM that contains one or more EFI drivers. The protocol instance is attached to the handle of the PCI controller.

EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL

Summary

This protocol matches one or more drivers to a controller. This protocol is produced by a bus driver, and it is installed on the child handles of buses that require a bus specific algorithm for matching drivers to controllers. This protocol is used by the **ConnectController()** boot service to select the best driver for a controller. All of the drivers returned by this protocol have a higher precedence than drivers found in the general EFI Driver Binding search algorithm, but a lower precedence than those drivers returned by the EFI Platform Driver Override Protocol. If more than one driver image handle is returned by this protocol, then the drivers image handles are returned in order from highest precedence to lowest precedence.

GUID

```
#define EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL_GUID \
  { 0x3bc1b285,0x8a15,0x4a82,0xaa,0xbf,0x4d,0x7d,0x13,0xfb,0x32,0x65 }
```

Protocol Interface Structure

```
typedef struct _EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL {
  EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_GET_DRIVER   GetDriver;
} EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL;
```

Parameters

GetDriver	Uses a bus specific algorithm to retrieve a driver image handle for a controller. See the **GetDriver()** function description.

Description

The **EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL** provides a mechanism for bus drivers to override the default driver selection performed by the **ConnectController()** boot service. This protocol is attached to the handle of a child device after the child handle is created by the bus driver. The service in this protocol can return a bus specific override driver to **ConnectController()**. **ConnectController()** must call this service until all of the bus specific override drivers have been retrieved. **ConnectController()** uses this information along with the EFI Platform Driver Override Protocol and all of the EFI Driver Binding protocol instances to select the best drivers for a controller. Since a controller can be managed by more than one driver, this protocol can return more than one bus specific override driver.

EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL.GetDriver()

Summary

Uses a bus specific algorithm to retrieve a driver image handle for a controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_GET_DRIVER) (
  IN     EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL *This,
  IN OUT EFI_HANDLE                                *DriverImageHandle
  );
```

Parameters

This	A pointer to the **EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL** instance.
DriverImageHandle	On input, a pointer to the previous driver image handle returned by **GetDriver()**. On output, a pointer to the next driver image handle. Passing in a **NULL**, will return the first driver image handle.

Description

This function is used to retrieve a driver image handle that is selected in a bus specific manner. The first driver image handle is retrieved by passing in a *DriverImageHandle* value of **NULL**. This will cause the first driver image handle to be returned in *DriverImageHandle*. On each successive call, the previous value of *DriverImageHandle* must be passed in. If a call to this function returns a valid driver image handle, then **EFI_SUCCESS** is returned. This process is repeated until **EFI_NOT_FOUND** is returned. If a *DriverImageHandle* is passed in that was not returned on a prior call to this function, then **EFI_INVALID_PARAMETER** is returned. The first driver image handle has the highest precedence, and the last driver image handle has the lowest precedence. This ordered list of driver image handles is used by the boot service **ConnectController()** to search for the best driver for a controller.

Status Codes Returned

EFI_SUCCESS	A bus specific override driver is returned in *DriverImageHandle*.
EFI_NOT_FOUND	The end of the list of override drivers was reached. A bus specific override driver is not returned in *DriverImageHandle*.
EFI_INVALID_PARAMETER	*DriverImageHandle* is not a handle that was returned on a previous call to **GetDriver()**.

9.4 EFI Driver Configuration Protocol

This section provides a detailed description of the **EFI_DRIVER_CONFIGURATION_ PROTOCOL**. This is a protocol that allows an EFI Driver to provide the ability to set controller specific options on a controller that the driver is managing. Unlike legacy option ROMs, the configuration of drivers and controllers is delayed until a platform management utility chooses to use the services of this protocol. EFI Drivers are not allowed to perform setup-like operations outside the context of this protocol. This means that a driver is not allowed to interact with the user outside the context of this protocol.

EFI_DRIVER_CONFIGURATION_PROTOCOL

Summary

Used to set configuration options for a controller that an EFI Driver is managing.

GUID

```
#define EFI_DRIVER_CONFIGURATION_PROTOCOL_GUID \
  { 0x107a772b,0xd5e1,0x11d4,0x9a,0x46,0x0,0x90,0x27,0x3f,0xc1,0x4d }
```

Protocol Interface Structure

```
typedef struct _EFI_DRIVER_CONFIGURATION_PROTOCOL {
  EFI_DRIVER_CONFIGURATION_SET_OPTIONS      SetOptions;
  EFI_DRIVER_CONFIGURATION_OPTIONS_VALID    OptionsValid;
  EFI_DRIVER_CONFIGURATION_FORCE_DEFAULTS   ForceDefaults;
  CHAR8                                     *SupportedLanguages;
} EFI_DRIVER_CONFIGURATION_PROTOCOL;
```

Parameters

SetOptions	Allows the use to set drivers specific configuration options for a controller that the driver is currently managing. See the **SetOptions()** function description.
OptionsValid	Tests to see if a controller's current configuration options are valid. See the **OptionsValid()** function description.
ForceDefaults	Forces a driver to set the default configuration options for a controller. See the **ForceDefaults()** function description.
SupportedLanguages	A Null-terminated ASCII string that contains one or more ISO 639-2 language codes. This is the list of language codes that this protocol supports.

Description

The **`EFI_DRIVER_CONFIGURATION_PROTOCOL`** is used by a platform management utility to allow the user to set controller specific options. This protocol is optionally attached to the image handle of driver in the driver's entry point. The platform management utility can collect all the **`EFI_DRIVER_CONFIGURATION_PROTOCOL`** instances present in the system, and present the user with a menu of the controllers than have user selectable options. This platform management utility is invoked through a platform component such as the EFI Boot Manager.

EFI_DRIVER_CONFIGURATION_PROTOCOL.SetOptions()

Summary

Allows the user to set controller specific options for a controller that a driver is currently managing.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DRIVER_CONFIGURATION_SET_OPTIONS) (
  IN  EFI_DRIVER_CONFIGURATION_PROTOCOL *This,
  IN  EFI_HANDLE                        ControllerHandle,
  IN  EFI_HANDLE                        ChildHandle  OPTIONAL,
  IN  CHAR8                             *Language,
  OUT EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED  *ActionRequired
  );
```

Parameters

This	A pointer to the **EFI_DRIVER_CONFIGURATION_PROTOCOL** instance.
ControllerHandle	The handle of the controller to set options on. If *ControllerHandle* is a valid **EFI_HANDLE** that is being managed by this driver, then the user will be allowed to set options for the controller specified by *ControllerHandle*. If this parameter is **NULL**, then the options will be set for all the controllers that this driver is currently managing. If *ControllerHandle* is **NULL**, then setting options for a child controller is not supported, so *ChildHandle* must also be **NULL**.
ChildHandle	The handle of the child controller to set options on. This is an optional parameter that may be **NULL**. It will be **NULL** for device drivers, and for a bus drivers that wish to set options for the bus controller. It will not be **NULL** for a bus driver that wishes to set options for one of its child controllers.
Language	A pointer to a three character ISO 639-2 language identifier. This is the language of the user interface that should be presented to the user, and it must match one of the languages specified in *SupportedLanguages*. The number of languages supported by a driver is up to the driver writer.
ActionRequired	A pointer to the action that the calling agent is required to perform when this function returns. See "Related Definitions" for a list of the actions that the calling agent is required to perform prior to accessing *ControllerHandle* again.

Description

This function allows the configuration options to be set for the driver specified by *This* on the controller specified by *ControllerHandle* and *ChildHandle*. This function must only use the **SIMPLE_INPUT_PROTOCOL** and **SIMPLE_TEXT_OUPUT_PROTOCOL** from the **EFI_SYSTEM_TABLE** to interact with the user, and it must use the language specified by *Language*. If the driver specified by *This* does not support the language specified by *Language*, then **EFI_UNSUPPORTED** is returned. If the controller specified by *ControllerHandle* and *ChildHandle* is not supported by the driver specified by *This*, then **EFI_UNSUPPORTED** is returned. If a device error occurs while setting the configuration options, **EFI_DEVICE_ERROR** is returned. If there are not enough resources available to set the configuration options, then **EFI_OUT_OF_RESOURCES** is returned.

The *ActionRequired* return value must always be set to a legal value by this function. The caller must perform the required action regardless of the return status. The calling agent must also perform the action described by *ActionRequired* prior to using any of the services produced by *ControllerHandle* or any of its children.

Related Definitions

```
//*******************************************************
// EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED
//*******************************************************
typedef enum {
  EfiDriverConfigurationActionNone                = 0,
  EfiDriverConfigurationActionStopController      = 1,
  EfiDriverConfigurationActionRestartController   = 2,
  EfiDriverConfigurationActionRestartPlatform     = 3,
  EfiDriverConfigurationActionMaximum
} EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED;
```

EfiDriverConfigurationActionNone

The controller specified by *ControllerHandle* is still in a usable state. No actions are required before this controller can be used again.

EfiDriverConfigurationStopController

The driver has detected that the controller specified by *ControllerHandle* is not in a usable state, and it needs to be stopped. The calling agent can use the **DisconnectController()** service to perform this operation, and it should be performed as soon as possible.

EfiDriverConfigurationRestartController

This controller specified by *ControllerHandle* needs to be stopped and restarted before it can be used again. The calling agent can use the **DisconnectController()** and **ConnectController()** services to perform this operation. The restart operation can be delayed until all of the configuration options have been set.

EfiDriverConfigurationRestartPlatform

A configuration change has been made that requires the platform to be restarted before the controller specified by *ControllerHandle* can be used again. The calling agent can use the **ResetSystem()** services to perform this operation. The restart operation can be delayed until all of the configuration options have been set.

Status Codes Returned

EFI_SUCCESS	The driver specified by *This* successfully set the configuration options for the controller specified by *ControllerHandle*.
EFI_INVALID_PARAMETER	*ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ChildHandle* is not **NULL** and it is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ActionRequired* is **NULL**.
EFI_UNSUPPORTED	The driver specified by *This* does not support setting configuration options for the controller specified by *ControllerHandle* and *ChildHandle*.
EFI_UNSUPPORTED	The driver specified by *This* does not support the language specified by *Language*.
EFI_DEVICE_ERROR	A device error occurred while attempt to set the configuration options for the controller specified by *ControllerHandle* and *ChildHandle*.
EFI_OUT_RESOURCES	There are not enough resources available to set the configuration options for the controller specified by *ControllerHandle* and *ChildHandle*.

EFI_DRIVER_CONFIGURATION_PROTOCOL.OptionsValid()

Summary

Tests to see if a controller's current configuration options are valid.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DRIVER_CONFIGURATION_OPTIONS_VALID) (
  IN  EFI_DRIVER_CONFIGURATION_PROTOCOL   *This,
  IN  EFI_HANDLE                          ControllerHandle,
  IN  EFI_HANDLE                          ChildHandle  OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_DRIVER_CONFIGURATION_PROTOCOL** instance.
ControllerHandle	The handle of the controller to test if it's current configuration options are valid.
ChildHandle	The handle of the child controller to test if it's current configuration options are valid. This is an optional parameter that may be **NULL**. It will be **NULL** for device drivers. It will also be **NULL** for a bus drivers that wish to test the configuration options for the bus controller. It will not be **NULL** for a bus driver that wishes to test configuration options for one of its child controllers.

Description

This function tests to see if the configuration options for the driver specified by *This* on the controller specified by *ControllerHandle* and *ChildHandle* are valid. If they are, then **EFI_SUCCESS** is returned. If they are not valid, then **EFI_DEVICE_ERROR** is returned. If the controller specified by *ControllerHandle* and *ChildHandle* is not currently being managed by the driver specified by *This*, then **EFI_UNSUPPORTED** is returned. This function is not allowed to interact with the user. Since the driver is responsible for maintaining the configuration options for each controller it manages, the exact method by which the configuration options are validated is driver specific.

Status Codes Returned

EFI_SUCCESS	The controller specified by *ControllerHandle* and *ChildHandle* that is being managed by the driver specified by *This* has a valid set of configuration options.
EFI_INVALID_PARAMETER	*ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ChildHandle* is not **NULL** and it is not a valid **EFI_HANDLE**.
EFI_UNSUPPORTED	The driver specified by *This* is not currently managing the controller specified by *ControllerHandle* and *ChildHandle*.
EFI_DEVICE_ERROR	The controller specified by *ControllerHandle* and *ChildHandle* that is being managed by the driver specified by *This* has an invalid set of configuration options.

EFI_DRIVER_CONFIGURATION_PROTOCOL.ForceDefaults()

Summary

Forces a driver to set the default configuration options for a controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DRIVER_CONFIGURATION_FORCE_DEFAULTS) (
  IN  EFI_DRIVER_CONFIGURATION_PROTOCOL     *This,
  IN  EFI_HANDLE                            ControllerHandle,
  IN  EFI_HANDLE                            ChildHandle   OPTIONAL,
  IN  UINT32                                DefaultType,
  OUT EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED   *ActionRequired
  );
```

Parameters

This	A pointer to the **EFI_DRIVER_CONFIGURATION_PROTOCOL** instance.
ControllerHandle	The handle of the controller to force default configuration options on.
ChildHandle	The handle of the child controller to force default configuration options on. This is an optional parameter that may be **NULL**. It will be **NULL** for device drivers. It will also be **NULL** for a bus drivers that wish to force default configuration options for the bus controller. It will not be **NULL** for a bus driver that wishes to force default configuration options for one of its child controllers.
DefaultType	The type of default configuration options to force on the controller specified by *ControllerHandle* and *ChildHandle*. See Table 9-1 for legal values. A *DefaultType* of 0x00000000 must be supported by this protocol.
ActionRequired	A pointer to the action that the calling agent is required to perform when this function returns. See "Related Definitions" in the **SetOptions()** function description for a list of the actions that the calling agent is required to perform prior to accessing *ControllerHandle* again.

Description

This function forces the default configuration options specified by *DefaultType* for the driver specified by *This* on the controller specified by *ControllerHandle* and *ChildHandle*. This function is not allowed to interact with the user. If the controller specified by *ControllerHandle* and *ChildHandle* is not supported by the driver specified by *This*, then **EFI_UNSUPPORTED** is returned. If the configuration type specified by *DefaultType* is not supported, then **EFI_UNSUPPORTED** is returned. If a device error occurs while setting the default configuration options, **EFI_DEVICE_ERROR** is returned. If there are not enough resources available to set the default configuration options, then **EFI_OUT_OF_RESOURCES** is returned.

The *ActionRequired* return value must always be set to a legal value by this function. The caller must perform the required action regardless of the return status. The calling agent must also perform the action described by *ActionRequired* prior to using any of the services produced by *ControllerHandle* or any of its children.

Table 9-1. EFI Driver Configuration Default Type

Bits	Description
Bit 0-15	If bits 16-31 are 0x0000, then the following values are defined:
0x0000	**Safe Defaults**. This type must be supported by all implementations of the EFI_DRIVER_CONFIGURATION_PROTOCOL. It places a controller a safe configuration that has the greatest probability of functioning correctly in a platform.
0x0001	**Manufacturing Defaults**. Optional type that places the controller in a configuration suitable for a manufacturing and test environment.
0x0002	**Custom Defaults**. Optional type that places the controller in a custom configuration.
0x0003	**Performance Defaults**. Optional type that places the controller in a configuration the maximizes the controller's performance in a platform.
	All other values are reserved for future versions of the EFI Specification.
Bits16-31	A value of 0x0000 is reserved by this specification. Values 0x0001-0xFFFF are available for expansion by third parties.

Status Codes Returned

EFI_SUCCESS	The driver specified by *This* successfully forced the default configuration options on the controller specified by *ControllerHandle* and *ChildHandle*.
EFI_INVALID_PARAMETER	*ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ChildHandle* is not **NULL** and it is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ActionRequired* is **NULL**.
EFI_UNSUPPORTED	The driver specified by *This* does not support forcing the default configuration options on the controller specified by *ControllerHandle* and *ChildHandle*.
EFI_UNSUPPORTED	The driver specified by This does not support the configuration type specified by *DefaultType*.
EFI_DEVICE_ERROR	A device error occurred while attempt to force the default configuration options on the controller specified by *ControllerHandle* and *ChildHandle*.
EFI_OUT_RESOURCES	There are not enough resources available to force the default configuration options on the controller specified by *ControllerHandle* and *ChildHandle*.

9.5 EFI Driver Diagnostics Protocol

This section provides a detailed description of the **EFI_DRIVER_DIAGNOSTICS_PROTOCOL**. This is a protocol that allows an EFI Driver to perform diagnostics on a controller that the driver is managing.

EFI_DRIVER_DIAGNOSTICS_PROTOCOL

Summary

Used to perform diagnostics on a controller that an EFI Driver is managing.

GUID

```
#define EFI_DRIVER_DIAGNOSTICS_PROTOCOL_GUID \
  { 0x0784924f,0xe296,0x11d4,0x9a,0x49,0x0,0x90,0x27,0x3f,0xc1,0x4d }
```

Protocol Interface Structure

```
typedef struct _EFI_DRIVER_DIAGNOSTICS_PROTOCOL {
  EFI_DRIVER_DIAGNOSTICS_RUN_DIAGNOSTICS  RunDiagnostics;
  CHAR8                                   *SupportedLanguages;
} EFI_DRIVER_DIAGNOSTICS_PROTOCOL;
```

Parameters

RunDiagnostics	Runs diagnostics on a controller. See the **RunDiagnostics()** function description.
SupportedLanguages	A Null-terminated ASCII string that contains one or more ISO 639-2 language codes. This is the list of language codes that this protocol supports.

Description

The **EFI_DRIVER_DIAGNOSTICS_PROTOCOL** is used by a platform management utility to allow the user to run driver specific diagnostics on a controller. This protocol is optionally attached to the image handle of driver in the driver's entry point. The platform management utility can collect all the **EFI_DRIVER_DISAGNOTICS_PROTOCOL** instances present in the system, and present the user with a menu of the controllers that have diagnostic capabilities. This platform management utility is invoked through a platform component such as the EFI Boot Manager.

EFI_DRIVER_DIAGNOSTICS_PROTOCOL.RunDiagnostics()

Summary

Runs diagnostics on a controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DRIVER_DIAGNOSTICS_RUN_DIAGNOSTICS) (
  IN  EFI_DRIVER_DIAGNOSTICS_PROTOCOL  *This,
  IN  EFI_HANDLE                       ControllerHandle,
  IN  EFI_HANDLE                       ChildHandle  OPTIONAL,
  IN  EFI_DRIVER_DIAGNOSTIC_TYPE       DiagnosticType,
  IN  CHAR8                            *Language,
  OUT EFI_GUID                         **ErrorType,
  OUT UINTN                            *BufferSize,
  OUT CHAR16                           **Buffer
  );
```

Parameters

This	A pointer to the **EFI_DRIVER_DIAGNOSTICS_PROTOCOL** instance.
ControllerHandle	The handle of the controller to run diagnostics on.
ChildHandle	The handle of the child controller to run diagnostics on. This is an optional parameter that may be **NULL**. It will be **NULL** for device drivers. It will also be **NULL** for a bus drivers that wish to run diagnostics on the bus controller. It will not be **NULL** for a bus driver that wishes to run diagnostics on one of its child controllers.
DiagnosticType	Indicates type of diagnostics to perform on the controller specified by *ControllerHandle* and *ChildHandle*. See "Related Definitions" for the list of supported types.
Language	A pointer to a three character ISO 639-2 language identifier. This is the language in which the optional error message should be returned in *Buffer*, and it must match one of the languages specified in *SupportedLanguages*. The number of languages supported by a driver is up to the driver writer.

ErrorType	A GUID that defines the format of the data returned in *Buffer*.
BufferSize	The size, in bytes, of the data returned in *Buffer*.
Buffer	A buffer that contains a Null-terminated Unicode string plus some additional data whose format is defined by *ErrorType*. *Buffer* is allocated by this function with **AllocatePool()**, and it is the caller's responsibility to free it with a call to **FreePool()**.

Description

This function runs diagnostics on the controller specified by *ControllerHandle* and *ChildHandle*. *DiagnoticType* specifies the type of diagnostics to perform on the controller specified by *ControllerHandle* and *ChildHandle*. If the driver specified by *This* does not support the language specified by *Language*, then **EFI_UNSUPPORTED** is returned. If the controller specified by *ControllerHandle* and *ChildHandle* is not supported by the driver specified by *This*, then **EFI_UNSUPPORTED** is returned. If the diagnostics type specified by *DiagnosticType* is not supported by this driver, then **EFI_UNSUPPORTED** is returned. If there are not enough resources available to complete the diagnostic, then **EFI_OUT_OF_RESOURCES** is returned. If the controller specified by *ControllerHandle* and *ChildHandle* passes the diagnostic, then **EFI_SUCCESS** is returned. Otherwise, **EFI_DEVICE_ERROR** is returned.

If the language specified by *Language* is supported by this driver, then status information is returned in *ErrorType*, *BufferSize*, and *Buffer*. *Buffer* contains a Null-terminated Unicode string followed by additional data whose format is defined by *ErrorType*. *BufferSize* is the size of *Buffer* is bytes, and it is the caller's responsibility to call **FreePool()** on *Buffer* when the caller is done with the return data. If there are not enough resources available to return the status information, then **EFI_OUT_OF_RESOURCES** is returned.

Related Definitions

```
//*******************************************************
// EFI_DRIVER_DIAGNOSTIC_TYPE
//*******************************************************
typedef enum {
  EfiDriverDiagnosticTypeStandard              = 0,
  EfiDriverDiagnosticTypeExtended              = 1,
  EfiDriverDiagnosticTypeManufacturing         = 2,
  EfiDriverDiagnosticTypeMaximum
} EFI_DRIVER_DIAGNOSTIC_TYPE;
```

`EfiDriverDiagnosticTypeStandard`

Performs standard diagnostics on the controller. This diagnostic type is required to be supported by all implementations of this protocol.

`EfiDriverDiagnosticTypeExtended`

This is an optional diagnostic type that performs diagnostics on the controller that may take an extended amount of time to execute.

`EfiDriverDiagnosticTypeManufacturing`

This is an optional diagnostic type that performs diagnostics on the controller that are suitable for a manufacturing and test environment.

Status Codes Returned

EFI_SUCCESS	The controller specified by *ControllerHandle* and *ChildHandle* passed the diagnostic.
EFI_INVALID_PARAMETER	*ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ChildHandle* is not **NULL** and it is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Language* is **NULL**.
EFI_INVALID_PARAMETER	*ErrorType* is **NULL**.
EFI_INVALID_PARAMETER	*BufferType* is **NULL**.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_UNSUPPORTED	The driver specified by *This* does not support running diagnostics for the controller specified by *ControllerHandle* and *ChildHandle*.
EFI_UNSUPPORTED	The driver specified by *This* does not support the type of diagnostic specified by *DiagnosticType*.
EFI_UNSUPPORTED	The driver specified by *This* does not support the language specified by *Language*.
EFI_OUT_OF_RESOURCES	There are not enough resources available to complete the diagnostics.
EFI_OUT_OF_RESOURCES	There are not enough resources available to return the status information in *ErrorType*, *BufferSize*, and *Buffer*.
EFI_DEVICE_ERROR	The controller specified by *ControllerHandle* and *ChildHandle* did not pass the diagnostic.

9.6 EFI Component Name Protocol

This section provides a detailed description of the **EFI_COMPONENT_NAME_PROTOCOL**. This is a protocol that allows an EFI Driver to provide a user readable name of an EFI Driver, and a user readable name for each of the controllers that the EFI Driver is managing. This protocol is used by platform management utilities that wish to display names of components. These names may include the names of expansion slots, external connectors, embedded devices, and add-in devices.

EFI_COMPONENT_NAME_PROTOCOL

Summary

Used to retrieve user readable names of EFI Drivers and controllers managed by EFI Drivers.

GUID

```
#define EFI_COMPONENT_NAME_PROTOCOL_GUID \
  { 0x107a772c,0xd5e1,0x11d4,0x9a,0x46,0x0,0x90,0x27,0x3f,0xc1,0x4d }
```

Protocol Interface Structure

```
typedef struct _EFI_COMPONENT_NAME_PROTOCOL {
  EFI_COMPONENT_NAME_GET_DRIVER_NAME       GetDriverName;
  EFI_COMPONENT_NAME_GET_CONTROLLER_NAME   GetControllerName;
  CHAR8                                    *SupportedLanguages;
} EFI_COMPONENT_NAME_PROTOCOL;
```

Parameters

GetDriverName	Retrieves a Unicode string that is the user readable name of the EFI Driver. See the **GetDriverName()** function description.
GetControllerName	Retrieves a Unicode string that is the user readable name of a controller that is being managed by an EFI Driver. See the **GetControllerName()** function description.
SupportedLanguages	A Null-terminated ASCII string that contains one or more ISO 639-2 language codes. This is the list of language codes that this protocol supports.

Description

The **EFI_COMPONENT_NAME_PROTOCOL** is used retrieve a driver's user readable name and the names of all the controllers that a driver is managing from the driver's point of view. Each of these names is returned as a Null-terminated Unicode string. The caller is required to specify the language in which the Unicode string is returned, and this language must be present in the list of languages that this protocol supports specified by *SupportedLanguages*.

EFI_COMPONENT_NAME_PROTOCOL.GetDriverName()

Summary

Retrieves a Unicode string that is the user readable name of the EFI Driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_COMPONENT_NAME_GET_DRIVER_NAME) (
  IN  EFI_COMPONENT_NAME_PROTOCOL *This,
  IN  CHAR8                       *Language,
  OUT CHAR16                      **DriverName
  );
```

Parameters

This	A pointer to the **EFI_COMPONENT_NAME_PROTOCOL** instance.
Language	A pointer to a three character ISO 639-2 language identifier. This is the language of the driver name that that the caller is requesting, and it must match one of the languages specified in *SupportedLanguages*. The number of languages supported by a driver is up to the driver writer.
DriverName	A pointer to the Unicode string to return. This Unicode string is the name of the driver specified by *This* in the language specified by *Language*.

Description

This function retrieves the user readable name of an EFI Driver in the form of a Unicode string. If the driver specified by *This* has a user readable name in the language specified by *Language*, then a pointer to the driver name is returned in *DriverName*, and **EFI_SUCCESS** is returned. If the driver specified by *This* does not support the language specified by *Language*, then **EFI_UNSUPPORTED** is returned.

Status Codes Returned

EFI_SUCCESS	The Unicode string for the user readable name in the language specified by *Language* for the driver specified by *This* was returned in *DriverName*.
EFI_INVALID_PARAMETER	*Language* is **NULL**.
EFI_INVALID_PARAMETER	*DriverName* is **NULL**.
EFI_UNSUPPORTED	The driver specified by *This* does not support the language specified by *Language*.

EFI_COMPONENT_NAME_PROTOCOL.GetControllerName()

Summary

Retrieves a Unicode string that is the user readable name of the controller that is being managed by an EFI Driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) (
  IN  EFI_COMPONENT_NAME_PROTOCOL *This,
  IN  EFI_HANDLE                  ControllerHandle,
  IN  EFI_HANDLE                  ChildHandle        OPTIONAL,
  IN  CHAR8                       *Language,
  OUT CHAR16                      **ControllerName
  );
```

Parameters

This	A pointer to the **EFI_COMPONENT_NAME_PROTOCOL** instance.
ControllerHandle	The handle of a controller that the driver specified by *This* is managing. This handle specifies the controller whose name is to be returned.
ChildHandle	The handle of the child controller to retrieve the name of. This is an optional parameter that may be **NULL**. It will be **NULL** for device drivers. It will also be **NULL** for a bus drivers that wish to retrieve the name of the bus controller. It will not be **NULL** for a bus driver that wishes to retrieve the name of a child controller.
Language	A pointer to a three character ISO 639-2 language identifier. This is the language of the controller name that that the caller is requesting, and it must match one of the languages specified in *SupportedLanguages*. The number of languages supported by a driver is up to the driver writer.
ControllerName	A pointer to the Unicode string to return. This Unicode string is the name of the controller specified by *ControllerHandle* and *ChildHandle* in the language specified by *Language* from the point of view of the driver specified by *This*.

Description

This function retrieves the user readable name of the controller specified by *ControllerHandle* and *ChildHandle* in the form of a Unicode string. If the driver specified by *This* has a user readable name in the language specified by *Language*, then a pointer to the controller name is returned in *ControllerName*, and **EFI_SUCCESS** is returned.

If the driver specified by *This* is not currently managing the controller specified by *ControllerHandle* and *ChildHandle*, then **EFI_UNSUPPORTED** is returned.

If the driver specified by *This* does not support the language specified by *Language*, then **EFI_UNSUPPORTED** is returned.

Status Codes Returned

EFI_SUCCESS	The Unicode string for the user readable name specified by *This*, *ControllerHandle*, *ChildHandle*, and *Language* was returned in *ControllerName*.
EFI_INVALID_PARAMETER	*ControllerHandle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*ChildHandle* is not **NULL** and it is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Language* is **NULL**.
EFI_INVALID_PARAMETER	*ControllerName* is **NULL**.
EFI_UNSUPPORTED	The driver specified by *This* is not currently managing the controller specified by *ControllerHandle* and *ChildHandle*.
EFI_UNSUPPORTED	The driver specified by *This* does not support the language specified by *Language*.

10
Protocols - Console Support

10.1 Console I/O Protocol

This chapter defines the Console I/O protocol. This protocol is used to handle input and output of text-based information intended for the system user during the operation of code in the EFI boot services environment. Also included here are the definitions of three console devices: one for input and one each for normal output and errors.

These interfaces are specified by function call definitions to allow maximum flexibility in implementation. For example, there is no requirement for compliant systems to have a keyboard or screen directly connected to the system. Implementations may choose to direct information passed using these interfaces in arbitrary ways provided that the semantics of the functions are preserved (in other words, provided that the information is passed to and from the system user).

10.1.1 Overview

The EFI console is built out of the **SIMPLE_INPUT** and **SIMPLE_TEXT_OUTPUT** protocols. These two protocols implement a basic text-based console that allows platform firmware, EFI applications, and EFI OS loaders to present information to and receive input from a system administrator. The EFI console consists of 16-bit Unicode characters, a simple set of input control characters (Scan Codes), and a set of output-oriented programmatic interfaces that give functionality equivalent to an intelligent terminal. The EFI console does not support pointing devices on input or bitmaps on output.

The EFI specification requires that the **SIMPLE_INPUT** protocol support the same languages as the corresponding **SIMPLE_TEXT_OUTPUT** protocol. The **SIMPLE_TEXT_OUTPUT** protocol is recommended to support at least the printable Basic Latin Unicode character set to enable standard terminal emulation software to be used with an EFI console. The Basic Latin Unicode character set implements a superset of ASCII that has been extended to 16-bit characters. Any number of other Unicode character sets may be optionally supported.

10.1.2 ConsoleIn Definition

The **SIMPLE_INPUT** protocol defines an input stream that contains Unicode characters and required EFI scan codes. Only the control characters defined in Table 10-1 have meaning in the Unicode input or output streams. The control characters are defined to be characters U+0000 through U+001F. The input stream does not support any software flow control.

Table 10-1. Supported Unicode Control Characters

Mnemonic	Unicode	Description
Null	U+0000	Null character ignored when received.
BS	U+0008	Backspace. Moves cursor left one column. If the cursor is at the left margin, no action is taken.
TAB	U+0x0009	Tab.
LF	U+000A	Linefeed. Moves cursor to the next line.
CR	U+000D	Carriage Return. Moves cursor to left margin of the current line.

The input stream supports Scan Codes in addition to Unicode characters. If the Scan Code is set to 0x00 then the Unicode character is valid and should be used. If the Scan Code is set to a non-0x00 value it represents a special key as defined by Table 10-2.

Table 10-2. EFI Scan Codes for `SIMPLE_INPUT_INTERFACE`

EFI Scan Code	Description
0x00	Null scan code.
0x01	Move cursor up 1 row.
0x02	Move cursor down 1 row.
0x03	Move cursor right 1 column.
0x04	Move cursor left 1 column.
0x05	Home.
0x06	End.
0x07	Insert.
0x08	Delete.
0x09	Page Up.
0x0a	Page Down.
0x0b	Function 1.
0x0c	Function 2.
0x0d	Function 3.
0x0e	Function 4.
0x0f	Function 5.
0x10	Function 6.
0x11	Function 7.
0x12	Function 8.
0x13	Function 9.
0x14	Function 10.
0x17	Escape.

10.2 Simple Input Protocol

The Simple Input protocol defines the minimum input required to support the *ConsoleIn* device.

SIMPLE_INPUT

Summary

This protocol is used to obtain input from the *ConsoleIn* device. The EFI specification requires that the **SIMPLE_INPUT** protocol support the same languages as the corresponding **SIMPLE_TEXT_OUTPUT** protocol.

GUID

```
#define SIMPLE_INPUT_PROTOCOL \
     { 387477c1-69c7-11d2-8e39-00a0c969723b }
```

Protocol Interface Structure

```
typedef struct _SIMPLE_INPUT_INTERFACE {
    EFI_INPUT_RESET                         Reset;
    EFI_INPUT_READ_KEY                      ReadKeyStroke;
    EFI_EVENT                               WaitForKey;
} SIMPLE_INPUT_INTERFACE;
```

Parameters

Reset	Reset the *ConsoleIn* device. See **Reset()**.
ReadKeyStroke	Returns the next input character. See **ReadKeyStroke()**.
WaitForKey	Event to use with **WaitForEvent()** to wait for a key to be available.

Description

The **SIMPLE_INPUT** protocol is used on the *ConsoleIn* device. It is the minimum required protocol for *ConsoleIn*.

SIMPLE_INPUT.Reset()

Summary

Resets the input device hardware.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_INPUT_RESET) (
      IN SIMPLE_INPUT_INTERFACE   *This,
      IN BOOLEAN                  ExtendedVerification
      );
```

Parameters

This	A pointer to the **SIMPLE_INPUT_INTERFACE** instance. Type **SIMPLE_INPUT_INTERFACE** is defined in Section 10.2
ExtendedVerification	Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description

The **Reset()** function resets the input device hardware.

As part of initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the *ExtendedVerification* flag is **TRUE** the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware and/or EFI driver to implement.

Status Codes Returned

EFI_SUCCESS	The device was reset.
EFI_DEVICE_ERROR	The device is not functioning correctly and could not be reset.

SIMPLE_INPUT.ReadKeyStroke()

Summary

Reads the next keystroke from the input device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_INPUT_READ_KEY) (
      IN SIMPLE_INPUT_INTERFACE   *This,
      OUT EFI_INPUT_KEY           *Key
      );
```

Parameters

This	A pointer to the **SIMPLE_INPUT_INTERFACE** instance. Type **SIMPLE_INPUT_INTERFACE** is defined in Section 10.2.
Key	A pointer to a buffer that is filled in with the keystroke information for the key that was pressed. Type **EFI_INPUT_KEY** is defined in "Related Definitions" below.

Related Definitions

```
//*******************************************************
// EFI_INPUT_KEY
//*******************************************************
typedef struct {
      UINT16     ScanCode;
      CHAR16     UnicodeChar;
} EFI_INPUT_KEY;
```

Description

The **`ReadKeyStroke()`** function reads the next keystroke from the input device. If there is no pending keystroke the function returns **`EFI_NOT_READY`**. If there is a pending keystroke, then *`ScanCode`* is the EFI scan code defined in Table 10-2. The *`UnicodeChar`* is the actual printable character or is zero if the key does not represent a printable character (control key, function key, etc.).

Status Codes Returned

EFI_SUCCESS	The keystroke information was returned.
EFI_NOT_READY	There was no keystroke data available.
EFI_DEVICE_ERROR	The keystroke information was not returned due to hardware errors.

10.2.1 ConsoleOut or StandardError

The **`SIMPLE_TEXT_OUTPUT`** protocol must implement the same Unicode code pages as the **`SIMPLE_INPUT`** protocol. The protocol must also support the Unicode control characters defined in Table 10-1. The **`SIMPLE_TEXT_OUTPUT`** protocol supports special manipulation of the screen by programmatic methods and therefore does not support the EFI scan codes defined in Table 10-2.

10.3 Simple Text Output Protocol

The Simple Text Output protocol defines the minimum requirements for a text-based *`ConsoleOut`* device. The EFI specification requires that the **`SIMPLE_INPUT`** protocol support the same languages as the corresponding **`SIMPLE_TEXT_OUTPUT`** protocol.

SIMPLE_TEXT_OUTPUT Protocol

Summary

This protocol is used to control text-based output devices.

GUID

```
#define SIMPLE_TEXT_OUTPUT_PROTOCOL \
     { 387477c2-69c7-11d2-8e39-00a0c969723b }
```

Protocol Interface Structure

```
typedef struct _SIMPLE_TEXT_OUTPUT_INTERFACE {
    EFI_TEXT_RESET                   Reset;
    EFI_TEXT_STRING                  OutputString;
    EFI_TEXT_TEST_STRING             TestString;
    EFI_TEXT_QUERY_MODE              QueryMode;
    EFI_TEXT_SET_MODE                SetMode;
    EFI_TEXT_SET_ATTRIBUTE           SetAttribute;
    EFI_TEXT_CLEAR_SCREEN            ClearScreen;
    EFI_TEXT_SET_CURSOR_POSITION     SetCursorPosition;
    EFI_TEXT_ENABLE_CURSOR           EnableCursor;
    SIMPLE_TEXT_OUTPUT_MODE          *Mode;
} SIMPLE_TEXT_OUTPUT_INTERFACE;
```

Parameters

`Reset`	Reset the *`ConsoleOut`* device. See **`Reset()`**.
`OutputString`	Displays the Unicode string on the device at the current cursor location. See **`OutputString()`**.
`TestString`	Tests to see if the *`ConsoleOut`* device supports this Unicode string. See **`TestString()`**.
`QueryMode`	Queries information concerning the output device's supported text mode. See **`QueryMode()`**.

SetMode	Sets the current mode of the output device. See **SetMode()**.
SetAttribute	Sets the foreground and background color of the text that is output. See **SetAttribute()**.
ClearScreen	Clears the screen with the currently set background color. See **ClearScreen()**.
SetCursorPosition	Sets the current cursor position. See **SetCursorPosition()**.
EnableCursor	Turns the visibility of the cursor on/off. See **EnableCursor()**.
Mode	Pointer to **SIMPLE_TEXT_OUTPUT_MODE** data. Type **SIMPLE_TEXT_OUTPUT_MODE** is defined in "Related Definitions" below.

The following data values in the **SIMPLE_TEXT_OUTPUT_MODE** interface are read-only and are changed by using the appropriate interface functions:

MaxMode	The number of modes supported by **QueryMode()** and **SetMode()**.
Mode	The text mode of the output device(s).
Attribute	The current character output attribute.
CursorColumn	The cursor's column.
CursorRow	The cursor's row.
CursorVisible	The cursor is currently visible or not.

Related Definitions

```
//*******************************************************
// SIMPLE_TEXT_OUTPUT_MODE
//*******************************************************
typedef struct {
    INT32                               MaxMode;
    // current settings
    INT32                               Mode;
    INT32                               Attribute;
    INT32                               CursorColumn;
    INT32                               CursorRow;
    BOOLEAN                             CursorVisible;
} SIMPLE_TEXT_OUTPUT_MODE;
```

Description

The **`SIMPLE_TEXT_OUTPUT`** protocol is used to control text-based output devices. It is the minimum required protocol for any handle supplied as the *`ConsoleOut`* or *`StandardError`* device. In addition, the minimum supported text mode of such devices is at least 80 x 25 characters.

A video device that only supports graphics mode is required to emulate text mode functionality. Output strings themselves are not allowed to contain any control codes other than those defined in Table 10-1. Positional cursor placement is done only via the **`SetCursorPosition()`** function. It is highly recommended that text output to the *`StandardError`* device be limited to sequential string outputs. (That is, it is not recommended to use **`ClearScreen()`** or **`SetCursorPosition()`** on output messages to *`StandardError`*.)

If the output device is not in a valid text mode at the time of the **`HandleProtocol()`** call, the device is to indicate that its *`CurrentMode`* is –1. On connecting to the output device the caller is required to verify the mode of the output device, and if it is not acceptable to set it to something it can use.

SIMPLE_TEXT_OUTPUT.Reset()

Summary

Resets the text output device hardware.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_TEXT_RESET) (
      IN SIMPLE_TEXT_OUTPUT_INTERFACE  *This,
      IN BOOLEAN                       ExtendedVerification
      );
```

Parameters

This	A pointer to the **SIMPLE_TEXT_OUTPUT_INTERFACE** instance. Type **SIMPLE_TEXT_OUTPUT_INTERFACE** is defined in the "Related Definitions" section of Section 10.3.
ExtendedVerification	Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description

The **Reset()** function resets the text output device hardware. The cursor position is set to (0, 0), and the screen is cleared to the default background color for the output device.

As part of initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the *ExtendedVerification* flag is **TRUE** the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware and/or EFI driver to implement.

Status Codes Returned

EFI_SUCCESS	The text output device was reset.
EFI_DEVICE_ERROR	The text output device is not functioning correctly and could not be reset.

SIMPLE_TEXT_OUTPUT.OutputString()

Summary

Writes a Unicode string to the output device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_TEXT_STRING) (
     IN SIMPLE_TEXT_OUTPUT_INTERFACE  *This,
     IN CHAR16                        *String
     );
```

Parameters

This	A pointer to the **SIMPLE_TEXT_OUTPUT_INTERFACE** instance. Type **SIMPLE_TEXT_OUTPUT_INTERFACE** is defined in the "Related Definitions" section of Section 10.3.
String	The Null-terminated Unicode string to be displayed on the output device(s). All output devices must also support the Unicode drawing characters defined in "Related Definitions."

Related Definitions

```
//*******************************************************
// UNICODE DRAWING CHARACTERS
//*******************************************************

#define BOXDRAW_HORIZONTAL                    0x2500
#define BOXDRAW_VERTICAL                      0x2502
#define BOXDRAW_DOWN_RIGHT                    0x250c
#define BOXDRAW_DOWN_LEFT                     0x2510
#define BOXDRAW_UP_RIGHT                      0x2514
#define BOXDRAW_UP_LEFT                       0x2518
#define BOXDRAW_VERTICAL_RIGHT                0x251c
#define BOXDRAW_VERTICAL_LEFT                 0x2524
#define BOXDRAW_DOWN_HORIZONTAL               0x252c
#define BOXDRAW_UP_HORIZONTAL                 0x2534
#define BOXDRAW_VERTICAL_HORIZONTAL           0x253c
```

```
#define BOXDRAW_DOUBLE_HORIZONTAL           0x2550
#define BOXDRAW_DOUBLE_VERTICAL             0x2551
#define BOXDRAW_DOWN_RIGHT_DOUBLE           0x2552
#define BOXDRAW_DOWN_DOUBLE_RIGHT           0x2553
#define BOXDRAW_DOUBLE_DOWN_RIGHT           0x2554
#define BOXDRAW_DOWN_LEFT_DOUBLE            0x2555
#define BOXDRAW_DOWN_DOUBLE_LEFT            0x2556
#define BOXDRAW_DOUBLE_DOWN_LEFT            0x2557

#define BOXDRAW_UP_RIGHT_DOUBLE             0x2558
#define BOXDRAW_UP_DOUBLE_RIGHT             0x2559
#define BOXDRAW_DOUBLE_UP_RIGHT             0x255a

#define BOXDRAW_UP_LEFT_DOUBLE              0x255b
#define BOXDRAW_UP_DOUBLE_LEFT              0x255c
#define BOXDRAW_DOUBLE_UP_LEFT              0x255d

#define BOXDRAW_VERTICAL_RIGHT_DOUBLE       0x255e
#define BOXDRAW_VERTICAL_DOUBLE_RIGHT       0x255f
#define BOXDRAW_DOUBLE_VERTICAL_RIGHT       0x2560

#define BOXDRAW_VERTICAL_LEFT_DOUBLE        0x2561
#define BOXDRAW_VERTICAL_DOUBLE_LEFT        0x2562
#define BOXDRAW_DOUBLE_VERTICAL_LEFT        0x2563

#define BOXDRAW_DOWN_HORIZONTAL_DOUBLE      0x2564
#define BOXDRAW_DOWN_DOUBLE_HORIZONTAL      0x2565
#define BOXDRAW_DOUBLE_DOWN_HORIZONTAL      0x2566

#define BOXDRAW_UP_HORIZONTAL_DOUBLE        0x2567
#define BOXDRAW_UP_DOUBLE_HORIZONTAL        0x2568
#define BOXDRAW_DOUBLE_UP_HORIZONTAL        0x2569

#define BOXDRAW_VERTICAL_HORIZONTAL_DOUBLE  0x256a
#define BOXDRAW_VERTICAL_DOUBLE_HORIZONTAL  0x256b
#define BOXDRAW_DOUBLE_VERTICAL_HORIZONTAL  0x256c

//*******************************************************
// EFI Required Block Elements Code Chart
//*******************************************************

#define BLOCKELEMENT_FULL_BLOCK             0x2588
#define BLOCKELEMENT_LIGHT_SHADE            0x2591
```

```
//*******************************************************
// EFI Required Geometric Shapes Code Chart
//*******************************************************

#define GEOMETRICSHAPE_UP_TRIANGLE          0x25b2
#define GEOMETRICSHAPE_RIGHT_TRIANGLE       0x25ba
#define GEOMETRICSHAPE_DOWN_TRIANGLE        0x25bc
#define GEOMETRICSHAPE_LEFT_TRIANGLE        0x25c4

//*******************************************************
// EFI Required Arrow shapes
//*******************************************************

#define ARROW_UP                           0x2191
#define ARROW_DOWN                         0x2193
```

Description

The `OutputString()` function writes a Unicode string to the output device. This is the most basic output mechanism on an output device. The *`String`* is displayed at the current cursor location on the output device(s) and the cursor is advanced according to the rules listed in Table 10-3.

Table 10-3. EFI Cursor Location/Advance Rules

Mnemonic	Unicode	Description
Null	U+0000	Ignore the character, and do not move the cursor.
BS	U+0008	If the cursor is not at the left edge of the display, then move the cursor left one column.
LF	U+000A	If the cursor is at the bottom of the display, then scroll the display one row, and do not update the cursor position. Otherwise, move the cursor down one row.
CR	U+000D	Move the cursor to the beginning of the current row.
Other	U+XXXX	Print the character at the current cursor position and move the cursor right one column. If this moves the cursor past the right edge of the display, then the line should wrap to the beginning of the next line. This is equivalent to inserting a CR and an LF. Note that if the cursor is at the bottom of the display, and the line wraps, then the display will be scrolled one line.

If desired, the system's NVRAM environment variables may be used at install time to determine the configured locale of the system or the installation procedure can query the user for the proper language support. This is then used to either install the proper EFI image/loader or to configure the installed image's strings to use the proper text for the selected locale.

Status Codes Returned

EFI_SUCCESS	The string was output to the device.
EFI_DEVICE_ERROR	The device reported an error while attempting to output the text.
EFI_UNSUPPORTED	The output device's mode is not currently in a defined text mode.
EFI_WARN_UNKNOWN_GLYPH	This warning code indicates that some of the characters in the Unicode string could not be rendered and were skipped.

SIMPLE_TEXT_OUTPUT.TestString()

Summary

Verifies that all characters in a Unicode string can be output to the target device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_TEXT_TEST_STRING) (
      IN SIMPLE_TEXT_OUTPUT_INTERFACE  *This,
      IN CHAR16                        *String
      );
```

Parameters

This	A pointer to the **SIMPLE_TEXT_OUTPUT_INTERFACE** instance. Type **SIMPLE_TEXT_OUTPUT_INTERFACE** is defined in the "Related Definitions" section of Section 10.3.
String	The Null-terminated Unicode string to be examined for the output device(s).

Description

The **TestString()** function verifies that all characters in a Unicode string can be output to the target device.

This function provides a way to know if the desired character set is present for rendering on the output device(s). This allows the installation procedure (or EFI image) to at least select a letter set that the output devices are capable of displaying. Since the output device(s) may be changed between boots, if the loader cannot adapt to such changes it is recommended that the loader call **OutputString()** with the text it has and ignore any "unsupported" error codes. The devices(s) that are capable of displaying the Unicode letter set will do so.

Status Codes Returned

EFI_SUCCESS	The device(s) are capable of rendering the output string.
EFI_UNSUPPORTED	Some of the characters in the Unicode string cannot be rendered by one or more of the output devices mapped by the EFI handle.

SIMPLE_TEXT_OUTPUT.QueryMode()

Summary

Returns information for an available text mode that the output device(s) supports.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_TEXT_QUERY_MODE) (
     IN SIMPLE_TEXT_OUTPUT_INTERFACE  *This,
     IN UINTN                         ModeNumber,
     OUT UINTN                        *Columns,
     OUT UINTN                        *Rows
     );
```

Parameters

This	A pointer to the **SIMPLE_TEXT_OUTPUT_INTERFACE** instance. Type **SIMPLE_TEXT_OUTPUT_INTERFACE** is defined in the "Related Definitions" section of Section 10.3.
ModeNumber	The mode number to return information on.
Columns, Rows	Returns the geometry of the text output device for the request *ModeNumber*.

Description

The **QueryMode()** function returns information for an available text mode that the output device(s) supports.

It is required that all output devices support at least 80x25 text mode. This mode is defined to be mode 0. If the output devices support 80x50, that is defined to be mode 1. Any other text dimensions supported by the device may then follow as mode 2 and above. (For example, it is a prerequisite that 80x25 and 80x50 text modes be supported before any other modes are.)

Status Codes Returned

EFI_SUCCESS	The requested mode information was returned.
EFI_DEVICE_ERROR	The device had an error and could not complete the request.
EFI_UNSUPPORTED	The mode number was not valid.

SIMPLE_TEXT_OUTPUT.SetMode()

Summary

Sets the output device(s) to a specified mode.

Prototype

```
EFI_STATUS
(* EFIAPI EFI_TEXT_SET_MODE) (
     IN SIMPLE_TEXT_OUTPUT_INTERFACE  *This,
     IN UINTN                          ModeNumber
     );
```

Parameters

This	A pointer to the **SIMPLE_TEXT_OUTPUT_INTERFACE** instance. Type **SIMPLE_TEXT_OUTPUT_INTERFACE** is defined in the "Related Definitions" section of Section 10.3.
ModeNumber	The text mode to set.

Description

The **SetMode()** function sets the output device(s) to the requested mode. On success the device is in the geometry for the requested mode, and the device has been cleared to the current background color with the cursor at (0,0).

Status Codes Returned

EFI_SUCCESS	The requested text mode was set.
EFI_DEVICE_ERROR	The device had an error and could not complete the request.
EFI_UNSUPPORTED	The mode number was not valid.

SIMPLE_TEXT_OUTPUT.SetAttribute()

Summary

Sets the background and foreground colors for the **OutputString()** and **ClearScreen()** functions.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_TEXT_SET_ATTRIBUTE) (
    IN SIMPLE_TEXT_OUTPUT_INTERFACE  *This,
    IN UINTN                         Attribute
    );
```

Parameters

This	A pointer to the **SIMPLE_TEXT_OUTPUT_INTERFACE** instance. Type **SIMPLE_TEXT_OUTPUT_INTERFACE** is defined in the "Related Definitions" section of Section 10.3.
Attribute	The attribute to set. Bits 0..3 are the foreground color, and bits 4..6 are the background color. All other bits are undefined and must be zero. See "Related Definitions" below.

Related Definitions

```
//*******************************************************
// Attributes
//*******************************************************
#define EFI_BLACK           0x00
#define EFI_BLUE            0x01
#define EFI_GREEN           0x02
#define EFI_CYAN            0x03
#define EFI_RED             0x04
#define EFI_MAGENTA         0x05
#define EFI_BROWN           0x06
#define EFI_LIGHTGRAY       0x07
#define EFI_BRIGHT          0x08
#define EFI_DARKGRAY        0x08
#define EFI_LIGHTBLUE       0x09
#define EFI_LIGHTGREEN      0x0A
#define EFI_LIGHTCYAN       0x0B
#define EFI_LIGHTRED        0x0C
#define EFI_LIGHTMAGENTA    0x0D
#define EFI_YELLOW          0x0E
#define EFI_WHITE           0x0F
```

```
#define EFI_BACKGROUND_BLACK          0x00
#define EFI_BACKGROUND_BLUE           0x10
#define EFI_BACKGROUND_GREEN          0x20
#define EFI_BACKGROUND_CYAN           0x30
#define EFI_BACKGROUND_RED            0x40
#define EFI_BACKGROUND_MAGENTA        0x50
#define EFI_BACKGROUND_BROWN          0x60
#define EFI_BACKGROUND_LIGHTGRAY      0x70

#define EFI_TEXT_ATTR(foreground,background)     \
      ((foreground) | ((background) << 4))
```

Description

The **SetAttribute()** function sets the background and foreground colors for the **OutputString()** and **ClearScreen()** functions.

The color mask can be set even when the device is in an invalid text mode.

Devices supporting a different number of text colors are required to emulate the above colors to the best of the device’s capabilities.

Status Codes Returned

EFI_SUCCESS	The requested attributes were set.
EFI_DEVICE_ERROR	The device had an error and could not complete the request.
EFI_UNSUPPORTED	The attribute requested is not defined by this specification.

SIMPLE_TEXT_OUTPUT.ClearScreen()

Summary

Clears the output device(s) display to the currently selected background color.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_TEXT_CLEAR_SCREEN) (
      IN SIMPLE_TEXT_OUTPUT_INTERFACE  *This
      );
```

Parameters

This	A pointer to the **SIMPLE_TEXT_OUTPUT_INTERFACE** instance. Type **SIMPLE_TEXT_OUTPUT_INTERFACE** is defined in the "Related Definitions" section of Section 10.3.

Description

The **ClearScreen()** function clears the output device(s) display to the currently selected background color. The cursor position is set to (0, 0).

Status Codes Returned

EFI_SUCCESS	The operation completed successfully.
EFI_DEVICE_ERROR	The device had an error and could not complete the request.
EFI_UNSUPPORTED	The output device is not in a valid text mode.

SIMPLE_TEXT_OUTPUT.SetCursorPosition()

Summary

Sets the current coordinates of the cursor position.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_TEXT_SET_CURSOR_POSITION) (
      IN SIMPLE_TEXT_OUTPUT_INTERFACE  *This,
      IN UINTN                         Column,
      IN UINTN                         Row
      );
```

Parameters

This	A pointer to the **SIMPLE_TEXT_OUTPUT_INTERFACE** instance. Type **SIMPLE_TEXT_OUTPUT_INTERFACE** is defined in the "Related Definitions" section of Section 10.3.
Column, Row	The position to set the cursor to. Must greater than or equal to zero and less than the number of columns and rows returned by **QueryMode()**.

Description

The **SetCursorPosition()** function sets the current coordinates of the cursor position. The upper left corner of the screen is defined as coordinate (0, 0).

Status Codes Returned

EFI_SUCCESS	The operation completed successfully.
EFI_DEVICE_ERROR	The device had an error and could not complete the request.
EFI_UNSUPPORTED	The output device is not in a valid text mode, or the cursor position is invalid for the current mode.

SIMPLE_TEXT_OUTPUT.EnableCursor()

Summary

Makes the cursor visible or invisible.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_TEXT_ENABLE_CURSOR) (
     IN SIMPLE_TEXT_OUTPUT_INTERFACE   *This,
     IN BOOLEAN                        Visible
     );
```

Parameters

This	A pointer to the **SIMPLE_TEXT_OUTPUT_INTERFACE** instance. Type **SIMPLE_TEXT_OUTPUT_INTERFACE** is defined in the "Related Definitions" section of Section 10.3.
Visible	If **TRUE**, the cursor is set to be visible. If **FALSE**, the cursor is set to be invisible.

Description

The **EnableCursor()** function makes the cursor visible or invisible.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully.
EFI_DEVICE_ERROR	The device had an error and could not complete the request or the device does not support changing the cursor mode.
EFI_UNSUPPORTED	The output device does not support visibility control of the cursor.

10.4 Universal Graphics Adapter Protocols

This section describes abstractions for displaying graphics on an EFI compliant platform. These abstractions consist of the UGA Draw Protocols that abstract the drawing to the graphics screen in the pre-OS space. The UGA I/O protocol also abstracts access to the graphics screen in addition to supporting child devices of the video controller, such as graphics display devices. The UGA I/O protocol is targeted primarily for use in the OS present environment.

The goal of this document is to replace the functionality that currently exists with VGA hardware and its corresponding video BIOS. The UGA ROM is a software abstraction and its goal is to support any foreseeable graphics hardware and not require VGA hardware, while at the same time also lending itself to implementation on the current generation of VGA hardware.

Graphics output is important in the preboot space to support modern firmware features. These features include the display of logos, the localization of output to any language, and setup and configuration screens.

There are also needs for graphics abstractions in a modern graphics oriented operating system. These operating systems generally contain a high performance driver that is specific to video device, but there are times when some required hardware related operations are not available in the high performance driver. In these cases it may be advantageous for the operating system to be able to take advantage of a graphics driver that was distributed with the graphics hardware that does include such capabilities.

More information on EFI 1.10 UGA ROM usage under an OS can be found at www.microsoft.com/hwdev/uga

10.4.1 UGA ROM

The **EFI_UGA_DRAW_PROTOCOL** provides a lightweight set of services to draw on a video screen. Graphics primitives are needed prior to operating system boot to support the localization of output to all known languages. The need for localization is the reason that **EFI_UGA_DRAW_PROTOCOL** does not support any text modes because a font database for all the glyphs in the Unicode character set would make an **EFI_UGA_DRAW_PROTOCOL** option ROM prohibitively large. The **EFI_UGA_DRAW_PROTOCOL** was constructed with the theory that the system firmware carries the fonts for the characters it chooses to display.

The availability of platform independent graphics primitives prior to an operating system boot allows the platform vendor to display a logo while the system is booting. Graphics primitives also allow more options for the user interfaces of configuration and diagnostic programs associated with the platform independent of the installed operating system.

The **EFI_UGA_IO_PROTOCOL** provides a mechanism for the OS to construct a generic OS specific driver that would make it possible to draw on an output device in the event that a high performance OS video driver was not available. The **EFI_UGA_IO_PROTOCOL** also provides services that can be used by an OS present high performance video driver.

10.4.2 UGA Draw Protocol

The **EFI_UGA_DRAW_PROTOCOL** supports three member functions to support the limited graphics needs of the pre-OS space. These member functions allow the caller to draw to a virtualized frame buffer, to get the current video mode, and to set a video mode. These simple primitives are sufficient to support the general needs of pre-OS firmware code

10.4.3 Blt Buffer

The basic graphics operation in the **EFI_UGA_DRAW_PROTOCOL** is the Block Transfer or Blt. The Blt operation allows data to be read or written to the video adapter's video memory. The Blt operation abstracts the video adapters hardware implementation by introducing the concept of a software Blt buffer.

The frame buffer abstracts the video display as an array of pixels. Each pixels location on the video display is defined by its X and Y coordinates. The X coordinate represents a scan line. A scan line is a horizontal line of pixels on the display. The Y coordinate represents a vertical line on the display. The upper left hand corner of the video display is defined as (0, 0) where the notation (X, Y) represents the X and Y coordinate of the pixel. The lower right corner of the video display is represented by (Width –1, Height -1).

The software Blt buffer is structured as an array of pixels. Pixel (0, 0) is the first element of the software Blt buffer. The Blt buffer can be thought of as a set of scan lines. It is possible to convert a pixel location on the video display to the Blt buffer using the following algorithm: Blt buffer array index = Y * Width + X.

Each software Blt buffer entry represents a pixel that is comprised of a 32-bit quantity. Byte zero of the Blt buffer entry represents the Red component of the pixel. Byte one of the Blt buffer entry represents the Green component of the pixel. Byte two of the Blt buffer entry represents the Blue component of the pixel. Byte three of the Blt buffer entry is reserved and must be zero.

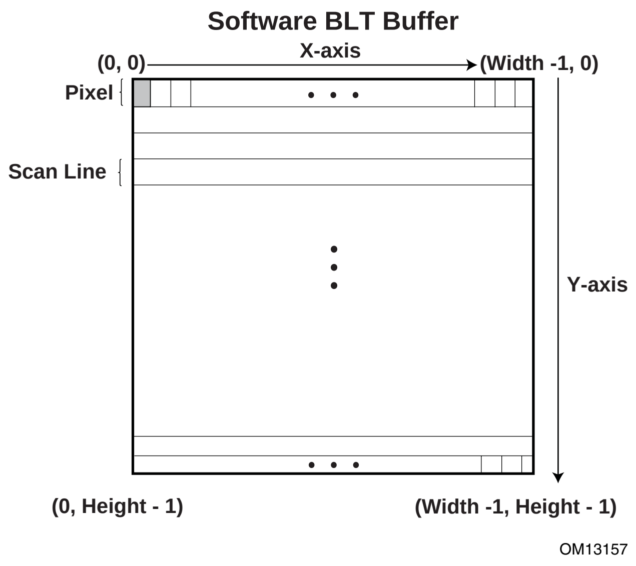


Figure 10-1. Software BLT Buffer

10.4.4 UGA I/O Protocol

The **EFI_UGA_IO_PROTOCOL** supports an I/O request mode of operation that is targeted at providing services to the OS high performance driver. The I/O requests are accessed via the **EFI_UGA_IO_PROTOCOL DispatchService()** member function. The I/O request services include the capabilities supported by the **EFI_UGA_DRAW_PROTOCOL**.

The I/O request mode services support a full set of services for the Graphics Controller, and all its child devices. Currently Output Controllers, Output Ports, and vendor defined child devices are supported. An example of an Output Controller would be a RAMDAC or TV OUT. An example of an Output Port would include a Monitor, TV, or HDTV.

10.4.5 Fallback Mode Driver

A fallback mode driver is defined as a simple driver that can be carried with a hardware device that can be made to run under any operating system given appropriate operating system support. The term fallback mode stems from the ability to use this driver when a standard high performance operating system driver is not available or some capability to manipulate the hardware is not available in the high performance operating system driver.

A fallback mode driver for a video device can be constructed by layering operating system specific code on top and below an EBC UGA option ROM. Since a UGA driver uses EFI abstractions to allocate memory and touch hardware it will be possible to replace the EFI 1.10 core services and protocols that a UGA ROM depends on with an operating specific driver. The operating system will also need to abstract the **EFI_UGA_IO_PROTOCOL** in some form that is compatible with the operating system driver model. The DispatchService member of the **EFI_UGA_IO_PROTOCOL** is designed to make binding this protocol into an Operation System device driver model.

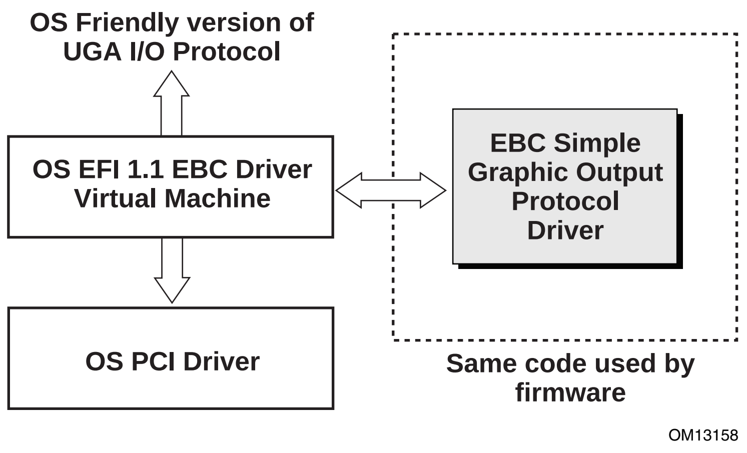

OS Friendly version of UGA I/O Protocol
OS EFI 1.1 EBC Driver Virtual Machine
EBC Simple Graphic Output Protocol Driver
OS PCI Driver
Same code used by firmware
OM13158

Figure 10-2. Fallback Mode Driver

From the UGA ROM's point of view it cannot tell the difference between a virtual machine that is produced in the pre-OS and OS present space.

10.5 UGA Draw Protocol

The interface structure for the UGA Draw Protocol is defined in this section. A unique UGA Draw Protocol must represent each video frame buffer in the system.

EFI_UGA_DRAW_PROTOCOL

Summary

Provides a basic abstraction to set video modes and copy pixels to and from the graphics controller's frame buffer.

GUID

```
#define EFI_UGA_DRAW_PROTOCOL_GUID \
  { 0x982c298b,0xf4fa,0x41cb,0xb8,0x38,0x77,0xaa,0x68,0x8f,0xb8,0x39 }
```

Protocol Interface Structure

```
typedef struct EFI_UGA_DRAW_PROTCOL {
  EFI_UGA_DRAW_PROTOCOL_GET_MODE        GetMode;
  EFI_UGA_DRAW_PROTOCOL_SET_MODE        SetMode;
  EFI_UGA_DRAW_PROTOCOL_BLT             Blt;
} EFI_UGA_DRAW_PROTOCOL;
```

Parameters

GetMode	Returns information about the geometry and configuration of the graphics controller's current frame buffer configuration.
SetMode	Set the graphics device into a given mode and clears the frame buffer to black.
Blt	Software abstraction to draw on the video device's frame buffer.

Description

The **EFI_UGA_DRAW_PROTOCOL** provides a software abstraction to allow pixels to be drawn directly to the frame buffer. The **EFI_UGA_DRAW_PROTOCOL** is designed to be lightweight and to support the basic needs of graphics output prior to Operating System boot.

A video device can support an arbitrary number of geometries, but it must support the following mode to conform to this specification:

- 800 x 600 with 32-bit color with a 60 Hz refresh rate.

EFI_UGA_DRAW_PROTOCOL.GetMode()

Summary

Return the current frame buffer geometry and display refresh rate.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_UGA_DRAW_PROTOCOL_GET_MODE) (
  IN  EFI_UGA_DRAW_PROTOCOL    *This,
  OUT UINT32                   *HorizontalResolution,
  OUT UINT32                   *VerticalResolution,
  OUT UINT32                   *ColorDepth,
  OUT UINT32                   *RefreshRate
  );
```

Parameters

This	The **EFI_UGA_DRAW_PROTOCOL** instance. Type **EFI_UGA_DRAW_PROTOCOL** is defined in Section 10.5.
HorizontalResolution	The size of video screen in pixels in the X dimension.
VerticalResolution	The size of video screen in pixels in the Y dimension.
ColorDepth	Number of bits per pixel, currently defined to be 32.
RefreshRate	The refresh rate of the monitor in Hertz.

Description

The **GetMode()** function returns information about the current mode. All UGA devices must support an 800 x 600 x 32-bit per pixel x 60 Hz mode of operation. A UGA device may support an arbitrary number of modes in addition to the required mode.

Status Codes Returned

EFI_SUCCESS	Valid mode information was returned.
EFI_DEVICE_ERROR	A hardware error occurred trying to retrieve the video mode.
EFI_INVALID_PARAMETER	*HorizontalResolution*, or *VerticalResolution*, or *RefreshRate*, is **NULL**.

EFI_UGA_DRAW_PROTOCOL.SetMode()

Summary

Set the video device into the specified mode and clears the output display to black.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_UGA_DRAW_PROTOCOL_SET_MODE) (
  IN  EFI_UGA_DRAW_PROTOCOL  *This,
  IN UINT32                   HorizontalResolution,
  IN UINT32                   VerticalResolution,
  IN UINT32                   ColorDepth,
  IN UINT32                   RefreshRate
  );
```

Parameters

This	The **EFI_UGA_DRAW_PROTOCOL** instance. Type **EFI_UGA_DRAW_PROTOCOL** is defined in Section 10.5.
HorizontalResolution	The size of video screen in pixels in the X dimension.
VerticalResolution	The size of video screen in pixels in the Y dimension.
ColorDepth	Number of bits per pixel, currently defined to be 32.
RefreshRate	The refresh rate of the monitor in Hertz.

Description

This **SetMode()** function sets the output device to the video mode specified by *HorizontalResolution*, *VerticalResolution*, and *RefreshRate*. If any of the arguments (*HorizontalResolution*, *VerticalResolution*, or *RefreshRate*) are not supported **EFI_UNSUPPORTED** is returned.

If a device error occurs while attempt to set the video mode, then **EFI_DEVICE_ERROR** is returned. On success the device is in the requested geometry and the hardware frame buffer has been cleared to black (Red = 0, Green = 0, Blue = 0) and any enabled video display device is updated.

Status Codes Returned

EFI_SUCCESS	Graphics mode was changed.
EFI_DEVICE_ERROR	The device had an error and could not complete the request.
EFI_UNSUPPORTED	*HorizontalResolution*, *VerticalResolution*, or *RefreshRate* is not supported.

EFI_UGA_DRAW_PROTOCOL.Blt()

Summary

Blt a rectangle of pixels on the graphics screen. Blt stands for BLock Transfer.

Prototype

```
typedef struct {
  UINT8   Blue;
  UINT8   Green;
  UINT8   Red;
  UINT8   Reserved;
} EFI_UGA_PIXEL;

typedef struct {
  EfiUgaVideoFill,
  EfiUgaVideoToBltBuffer,
  EfiUgaBltBufferToVideo,
  EfiUgaVideoToVideo,
  EfiUgaBltMax
} EFI_UGA_BLT_OPERATION;

typedef
EFI_STATUS
(EFIAPI *EFI_UGA_DRAW_PROTOCOL_BLT) (
  IN EFI_UGA_DRAW_PROTOCOL     *This,
  IN OUT EFI_UGA_PIXEL          *BltBuffer,  OPTIONAL
  IN  EFI_UGA_BLT_OPERATION     BltOperation,
  IN UINTN                      SourceX,
  IN UINTN                      SourceY,
  IN UINTN                      DestinationX,
  IN UINTN                      DestinationY,
  IN UINTN                      Width,
  IN UINTN                      Height,
  IN UINTN                       Delta       OPTIONAL
  );
```

Parameters

This	The **EFI_UGA_DRAW_PROTOCOL** instance.
BltBuffer	The data to transfer to the graphics screen. Size is at least *Width***Height**sizeof(**EFI_UGA_PIXEL**).
BltOperation	The operation to perform when copying *BltBuffer* on to the graphics screen.
SourceX	The X coordinate of the source for the *BltOperation*. The origin of the screen is 0, 0 and that is the upper left-hand corner of the screen.
SourceY	The Y coordinate of the source for the *BltOperation*. The origin of the screen is 0, 0 and that is the upper left-hand corner of the screen.
DestinationX	The X coordinate of the destination for the *BltOperation*. The origin of the screen is 0, 0 and that is the upper left-hand corner of the screen.
DestinationY	The Y coordinate of the destination for the *BltOperation*. The origin of the screen is 0, 0 and that is the upper left-hand corner of the screen.
Width	The width of a rectangle in the blt rectangle in pixels. Each pixel is represented by an **EFI_UGA_PIXEL** element.
Height	The height of a rectangle in the blt rectangle in pixels. Each pixel is represented by an **EFI_UGA_PIXEL** element.
Delta	Not used for *EfiUgaVideoFill* or the *EfiUgaVideoToVideo* operation. If a *Delta* of zero is used, the entire *BltBuffer* is being operated on. If a subrectangle of the *BltBuffer* is being used then *Delta* represents the number of bytes in a row of the *BltBuffer*.

Description

The `Blt()` function is used to draw the *BltBuffer* rectangle onto the video screen.

The *BltBuffer* represents a rectangle of *Height* by *Width* pixels that will be drawn on the graphics screen using the operation specified by *BltOperation*. The *Delta* value can be used to enable the *BltOperation* to be performed on a sub-rectangle of the *BltBuffer*.

Table 10-4 describes the *BltOperation*s that are supported on rectangles. Rectangles have coordinates (left, upper) (right, bottom):

Table 10-4. Blt Operation Table

Blt Operation	Operation
EfiUgaVideoFill	Write data from the *BltBuffer* pixel (*SourceX*, *SourceY*) directly to every pixel of the video display rectangle (*DestinationX*, *DestinationY*) (*DestinationX* + *Width*, *DestinationY* + *Height*). Only one pixel will be used from the *BltBuffer*. *Delta* is NOT used.
EfiUgaVideoToBltBuffer	Read data from the video display rectangle (*SourceX*, *SourceY*) (*SourceX* + *Width*, *SourceY* + *Height*) and place it in the *BltBuffer* rectangle (*DestinationX*, *DestinationY*) (*DestinationX* + *Width*, *DestinationY* + *Height*). If *DestinationX* or *DestinationY* is not zero then *Delta* must be set to the length in bytes of a row in the *BltBuffer*.
EfiUgaBltBufferToVideo	Write data from the *BltBuffer* rectangle (*SourceX*, *SourceY*) (*SourceX* + *Width*, *SourceY* + *Height*) directly to the video display rectangle (*DestinationX*, *DestinationY*) (*DestinationX* + *Width*, *DestinationY* + *Height*). If *SourceX* or *SourceY* is not zero then *Delta* must be set to the length in bytes of a row in the *BltBuffer*.
EfiUgaVideoToVideo	Copy from the video display rectangle (*SourceX*, *SourceY*) (*SourceX* + *Width*, *SourceY* + *Height*) to the video display rectangle (*X*, *Y*) (*X* + *Width*, *Y* + *Height*). The *BltBuffer* and *Delta* are not used in this mode. There is no limitation on the overlapping of the source and destination rectangles.

Status Codes Returned

EFI_SUCCESS	*BltBuffer* was drawn to the graphics screen.
EFI_INVALID_PARAMETER	*BltOperation* is not valid.
EFI_DEVICE_ERROR	The device had an error and could not complete the request.

10.6 Rules for PCI/AGP Devices

In an EFI system that contains PCI or AGP devices each PCI device/function will be abstracted by a PCI I/O protocol on a handle with its associated device path.

If the PCI device/function contains a single frame buffer the **EFI_UGA_DRAW_PROTOCOL** must be placed on the same handle as the PCI I/O Protocol.

If the PCI device/function contains multiple frame buffers the UGA ROM must create child handles of the PCI I/O protocol that inherit its device path and append a controller device path node. The UGA ROM is responsible for creating the child handle and placing the device path protocol and **EFI_UGA_DRAW_PROTOCOL**.

10.7 UGA I/O Protocol

The interface structure for the UGA I/O Protocol is defined in this section. Each device abstracted by a UGA ROM must produce a unique **EFI_UGA_IO_PROTOCOL**.

EFI_UGA_IO_PROTOCOL

Summary

Provides a basic abstraction to send I/O requests to the graphics device and any of its children.

GUID

```
#define EFI_UGA_IO_PROTOCOL_GUID \
  { 0x61a4d49e, 0x6f68, 0x4f1b, 0xb9,0x22,0xa8,0x6e,0xed,0xb,0x7,0xa2}
```

Protocol Interface Structure

```
typedef struct {
  EFI_UGA_IO_PROTOCOL_CREATE_DEVICE     CreateDevice;
  EFI_UGA_IO_PROTOCOL_DELETE_DEVICE     DeleteDevice;
  PUGA_FW_SERVICE_DISPATCH              DispatchService;
} EFI_UGA_IO_PROTOCOL;
```

Parameters

CreateDevice	Create a **UGA_DEVICE** object for a child device of a given parent **UGA_DEVICE**.
DeleteDevice	Delete the **UGA_DEVICE** returned from **CreateDevice()**.
DispatchService	Dispatches I/O requests to the display device and its associate child devices.

Description

The **EFI_UGA_IO_PROTOCOL** is the primary interface exported by a UGA ROM in the OS present environment. The **EFI_UGA_IO_PROTOCOL.DispatchService()** allows communication with the video frame buffer and all its associated child devices. Child devices of the **EFI_UGA_IO_PROTOCOL** include output controllers such as a TV tuner, and display devices such as a HDTV.

The **EFI_UGA_IO_PROTOCOL** operates on **UGA_DEVICE** objects. Child devices can be enumerated by using **DispatchService()** to send a *pIoRequest* of type *UgaIoGetChildDevice*. A **UGA_DEVICE** object can be created via a call to **CreateDevice()** with the data returned from *UgaIoGetChildDevice*.

A video device can support an arbitrary number of geometries, but it must support one of the following modes to operate with the **EFI_UGA_IO_PROTOCOL**:

- 800 x 600 with 32-bit color with a 60 Hz refresh rate.

The advanced features of a UGA device are accessible via its **DispatchService()**. More information on the advanced capabilities of an EFI 1.10 UGA ROM can be found at www.microsoft.com/hwdev/uga.

EFI_UGA_IO_PROTOCOL.CreateDevice()

Summary

Dynamically allocate storage for a child **UGA_DEVICE**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_UGA_IO_CREATE_DEVICE) (
  IN  EFI_UGA_IO_PROTOCOL   *This,
  IN  UGA_DEVICE            *ParentDevice,
  IN  UGA_DEVICE_DATA       *DeviceData,
  IN  VOID                  *RunTimeContext,
  OUT UGA_DEVICE            **Device
  );
```

Parameters

This	The **EFI_UGA_IO_PROTOCOL** instance. Type **EFI_UGA_IO_PROTOCOL** is defined in Section 10.7.
ParentDevice	*ParentDevice* specifies a pointer to the parent device of *Device*.
DeviceData	A pointer to **UGA_DEVICE_DATA** returned from a call to **DispatchService()** with a **UGA_DEVICE** of *Parent* and an *IoRequest* of type *UgaIoGetChildDevice*.
RuntimeContext	Context to associate with *Device*.
Device	The *Device* returns a dynamically allocated child **UGA_DEVICE** object for *ParentDevice*. The caller is responsible for deleting *Device*.

Description

A **UGA_DEVICE** object contains data fields that are defined by this specification and pointers to implementation specific data structures. Since a **UGA_DEVICE** contains implementation specific data that must be dynamically allocated, the **CreateDevice()** member function is required to create a **UGA_DEVICE** object to enable the enumerate all the child **UGA_DEVICE**(s).

The device must not be started when its **UGA_DEVICE** *Device* is allocated.

Status Codes Returned

EFI_SUCCESS	*Device* was returned
EFI_INVALID_PARAMETER	One of the arguments was not valid
EFI_DEVICE_ERROR	The device had an error and could not complete the request.

EFI_UGA_IO_PROTOCOL.DeleteDevice()

Summary

Deletes a dynamically allocated child **UGA_DEVICE** object that was allocated using **CreateDevice()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_UGA_IO_DELETE_DEVICE) (
  IN EFI_UGA_IO_PROTOCOL *This,
  IN UGA_DEVICE          *Device
  );
```

Parameters

This	The **EFI_UGA_IO_PROTOCOL** instance. Type **EFI_UGA_IO_PROTOCOL** is defined in Section 10.7.
Device	The *Device* points to a **UGA_DEVICE** object that was dynamically allocated via a **CreateDevice()** call.

Description

An object that was created via a **CreateDevice()** is destroyed.

Status Codes Returned

EFI_SUCCESS	The *Device* was deleted.
EFI_INVALID_PARAMETER	The *Device* was not allocated via **CreateDevice()**.

PUGA_FW_SERVICE_DISPATCH.DispatchService()

Summary

This function is the main UGA service dispatch routine for all **UGA_IO_REQUEST**s.

Prototype

```
typedef
UGA_STATUS
(EFIAPI *PUGA_FW_SERVICE_DISPATCH) (
  IN  PUGA_DEVICE             pDevice,
  IN OUT PUGA_IO_REQUEST      pIoRequest
  );
```

Parameters

pDevice	*pDevice* specifies a pointer to a device object associated with a device enumerated by a *pIoRequest->ioRequestCode* of type *UgaIoGetChildDevice*. The root device for the **EFI_UGA_IO_PROTOCOL** is represented by *pDevice* being set to **NULL**.
pIoRequest	*pIoRequest* points to a caller allocated buffer that contains data defined by *pIoRequest->ioRequestCode*. See "Related Definitions" below for a definition of **UGA_IO_REQUEST_CODE**s and their associated data structures.

Related Definitions

```
typedef UINT32  UGA_STATUS;

typedef enum _UGA_DEVICE_TYPE {
    UgaDtParentBus = 1,
    UgaDtGraphicsController,
    UgaDtOutputController,
    UgaDtOutputPort,
    UgaDtOther
} UGA_DEVICE_TYPE, *PUGA_DEVICE_TYPE;

typedef UINT32 UGA_DEVICE_ID, *PUGA_DEVICE_ID;

typedef struct _UGA_DEVICE_DATA {
    UGA_DEVICE_TYPE deviceType;
    UGA_DEVICE_ID   deviceId;
    UINT32          ui32DeviceContextSize;
    UINT32          ui32SharedContextSize;
} UGA_DEVICE_DATA, *PUGA_DEVICE_DATA;
```

```
typedef struct _UGA_DEVICE {
    PVOID                   pvDeviceContext;
    PVOID                   pvSharedContext;
    PVOID                   pvRunTimeContext;
    struct _PUGA_DEVICE pParentDevice;
    PVOID                   pvBusIoServices;
    PVOID                   pvStdIoServices;
    UGA_DEVICE_DATA         deviceData;
} UGA_DEVICE, *PUGA_DEVICE;
```

A **UGA_DEVICE** is the basic device abstraction for enumerating child devices behind an **EFI_UGA_IO_PROTOCOL**. A **UGA_DEVICE** object is allocated dynamically via a call to *CreateDevice()*. A programmatic abstraction is required to allocate a **UGA_DEVICE**, since some of the data structures pointed to by a **UGA_DEVICE** are implementation specific.

```
typedef enum _UGA_IO_REQUEST_CODE {
    UgaIoGetVersion = 1,
    UgaIoGetChildDevice,
    UgaIoStartDevice,
    UgaIoStopDevice,
    UgaIoFlushDevice,
    UgaIoResetDevice,
    UgaIoGetDeviceState,
    UgaIoSetDeviceState,
    UgaIoSetPowerState,
    UgaIoGetMemoryConfiguration,
    UgaIoSetVideoMode,
    UgaIoCopyRectangle,
    UgaIoGetEdidSegment,
    UgaIoDeviceChannelOpen,
    UgaIoDeviceChannelClose,
    UgaIoDeviceChannelRead,
    UgaIoDeviceChannelWrite,
    UgaIoGetPersistentDataSize,
    UgaIoGetPersistentData,
    UgaIoSetPersistentData,
    UgaIoGetDevicePropertySize,
    UgaIoGetDeviceProperty,
    UgaIoBtPrivateInterface
} UGA_IO_REQUEST_CODE, *PUGA_IO_REQUEST_CODE;
```

```
typedef struct _UGA_IO_REQUEST {
  IN UGA_IO_REQUEST_CODE ioRequestCode;
  IN PVOID               pvInBuffer;
  IN UINT64              ui64InBufferSize;
  OUT PVOID              pvOutBuffer;
  IN UINT64              ui64OutBufferSize;
  OUT UINT64             ui64BytesReturned;
} UGA_IO_REQUEST, *PUGA_IO_REQUEST;
```

A more complete definition of the data structures in this section can be found at www.microsoft.com/hwdev/uga

Description

This is the main UGA service dispatch routine for all **UGA_IO_REQUEST**s. The **DispatchService()** method exports all the support **UGA_IO_REQUEST** firmware functionality of a device.

The **EFI_UGA_DRAW_PROTCOL** exist to provide lightweight access methods in the pre-OS space to draw on the video screen. All the functionality of **EFI_UGA_DRAW_PROTCOL** can be accessed directly via an *IoRequest*.

Status Codes Returned

The status returned by this function is defined on a per *IoRequest* basis.

10.8 Implementation Rules for an EFI UGA Driver

An EFI driver designed to manage the UGA controller must follow the EFI 1.10 driver model and thus produce an **EFI_DRIVER_BINDING_PROTOCOL** and follow the rules on implementing the **Supported()**, **Start()**, and **Stop()**. The **Start()** function must not initialize or start the video hardware, and it should just register an **EFI_UGA_IO_PROTOCOL** and one or more **EFI_UGA_DRAW_PROTOCOL**(s). The video hardware must be initialized via **EFI_UGA_IO_PROTOCOL** I/O requests or via the first call to **EFI_UGA_DRAW_PROTOCOL.SetMode()**.

An **EFI_UGA_DRAW_PROTOCOL** must be implemented for every video frame buffer that exists on a video adapter. In most cases there will be a single **EFI_UGA_DRAW_PROTOCOL** placed on the Controller handle passed into the **EFI_DRIVER_BINDING.Start()** function. As a UGA ROM can contain more than one EFI Image, the **EFI_UGA_DRAW_PROTOCOL** can be produced by a separate driver that consumes the **EFI_UGA_IO_PROTOCOL**.

An **EFI_UGA_IO_PROTOCOL** must be produced on the Controller handle passed into the **EFI_DRIVER_BINDING.Start()** function. There is only one **EFI_UGA_IO_PROTOCOL** produced for every device being managed by an UGA ROM.

For PCI based video device all hardware access will be done via **EFI_UGA_IO_PROTOCOL**. This includes IO, MMIO, BAR based access, and DMA.

The EFI Boot Service and Runtime APIs are used to allocate memory and register protocol interfaces.

Every UGA device must support an 800 x 600 x 32-bit color per pixel at 60 Hz by video mode.

The **EFI_UGA_IO_PROTOCOL.UgaIoDispatchServce()** function must support the following **UGA_IO_REQUEST**s:

```
UgaIoCopyRectangle
UgaIoFlushDevice
UgaIoGetChildDevice
UgaIoGetDeviceProperty
UgaIoGetDevicePropertySize
UgaIoGetDeviceState
UgaIoGetMemoryConfiguration
UgaIoResetDevice
UgaIoSetDeviceState
UgaIoSetPowerState
UgaIoSetVideoMode
UgaIoStartDevice
UgaIoStopDevice
```

The following **UGA_IO_REQUEST**s may not be required for specific hardware configurations:

```
UgaIoDeviceChannelClose
UgaIoDeviceChannelOpen
UgaIoDeviceChannelRead
UgaIoDeviceChannelWrite
UgaIoGetEdidSegment
```

For additional information on how implementations can be constructed please refer to the specification found at www.microsoft.com/hwdev/uga

10.9 UGA Draw Protocol to UGA I/O Protocol Mapping

As the **EFI_UGA_DRAW_PROTOCOL** member functions, **GetMode()**, **SetMode()**, and **Blt()** exist as a lightweight abstraction of the more extensive functionality abstracted by **DispatchService()**. This section describes the conceptual relationship between the protocol member functions and I/O requests.

The **GetMode()** function can be implemented via remembering the values passed to the previous call to **SetMode()**.

The **SetMode()** function can be implemented via an `UgaIoSetVideoMode` I/O request. The actual geometry of screen can be read via an `UgaIoGetMemoryConfiguration` I/O request.

The **Blt()** function can be implemented via an *`UgaIoCopyRectangle`* I/O request.

10.9.1 UGA System Requirements

This section defines the requirements a system must meet to support an EFI 1.10 UGA driver. A system could be defined as an EFI firmware implementation or a Virtual Machine (VM) that runs under an OS.

10.9.2 System Abstraction Requirements

The system must support the loading of an EFI 1.10 image. The system must support the EBC image type, and it may optionally support native images. When an EFI 1.10 driver is started it is passed a pointer to the EFI 1.10 System Table, and an EFI Image Handle for the loaded image. Thus the system must support the EFI system table and its associated runtime and boot services.

For PCI or AGP devices the system must produce a PCI_IO protocol on a handle for every UGA device that can be supported.

The system will follow the following sequence of events to bind an EFI UGA driver to a hardware device:

1. Initialize the EFI firmware or VM.
2. Create handles and PCI_IO protocols to abstract the supported devices.
3. Load the EFI 1.10 UGA drivers (drivers register Driver Binding Protocol but do not touch hardware).
4. Bind the EFI 1.10 UGA driver to the hardware device. EFI firmware or VM uses *gBS->*`ConnectController()` to bind driver handle to the PCI_IO device handle.
5. UGA protocols are now available for use.

10.9.3 Firmware to OS Hand-off

The system firmware must hand off to the OS the devices to which EFI 1.10 ROMs should be bound. The EFI firmware must create entries in the Configuration Table of the EFI System Table.

The Configuration Table entry for EFI 1.10 UGA ROMs will contain the **`EFI_UGA_IO_PROTOCOL_GUID`** and a pointer to the **`EFI_DRIVER_OS_HANDOFF_HEADER`** (See "Related Definitions" below.) The **`EFI_DRIVER_OS_HANDOFF_HEADER`** describes a list of **`EFI_DRIVER_OS_HANDOFF`** structures that describe to the OS what EFI 1.10 UGA ROMs are present in the system.

There is an **`EFI_DRIVER_OS_HANDOFF`** entry for each PCI device that the firmware discovered that is capable of supporting UGA. There may also be **`EFI_DRIVER_OS_HANDOFF`** entries for EFI 1.10 UGA drivers that were not associated with a device. It should be noted that the **`PciRomImage`** for a device may not contain the **`PeImage`** that firmware used as an EFI 1.10 UGA driver for the device.

Related Definitions

```
typedef struct {
  UINT32        Version;
  UINT32        HeaderSize;
  UINT32        SizeOfEntries;
  UINT32        NumberOfEntries;
} EFI_DRIVER_OS_HANDOFF_HEADER;
```

```
typedef enum {
  EfiUgaDriverFromPciRom,
  EfiUgaDriverFromSystem,
  EfiDriverHandoffMax
} EFI_DRIVER_HANOFF_ENUM;

typedef struct {
  EFI_DRIVER_HANOFF_ENUM Type;
  EFI_DEVICE_PATH        *DevicePath;
  VOID                   *PciRomImage;
  UINT64                 PciRomSize;
} EFI_DRIVER_OS_HANDOFF;
```

Type	The type of the **EFI_DRIVER_OS_HANDOFF** structure. Currently only **EfiUgaDriverHandoff** is defined and it represents the **EFI_DRIVER_OS_HANDOFF** in the context of a UGA device.
DevicePath	Pointer to the EFI device path that represents the UGA PCI address. Please note the device path does not contain the PCI bus as it may change from boot to boot.
PciRomImage	If *Type* is **EfiUgaDriverFromPciRom** then *PciRomImage* represents the contains of the PCI devices ROM bar. If *Type* is **EfiUgaDriverFromSystem** then the *PciRomImage* was produced by system for an onboard device. A PCI ROM can contain multiple EFI images and every image in the ROM must be loaded.
PciRomSize	The size of *PciRomImage* in bytes. The size will only include areas defines in the PCI 2.2 Option ROM header and not the entire space decoded by the ROM BAR. For example if the ROM BAR decoded to 16 MB, but the ROM image physically only contained 64 KB of information this value would be 64 KB.

10.10 Simple Pointer Protocol

This section defines the Simple Pointer Protocol and a detailed description of the **EFI_SIMPLE_POINTER_PROTOCOL**. The intent of this section is to specify a simple method for accessing pointer devices. This would include devices such as mice and trackballs.

The **EFI_SIMPLE_POINTER_PROTOCOL** allows information about a pointer device to be retrieved. This would include the status of buttons and the motion of the pointer device since the last time it was accessed. This protocol is attached the device handle of a pointer device, and can be used for input from the user in the preboot environment.

EFI_SIMPLE_POINTER_PROTOCOL

Summary

Provides services that allow information about a pointer device to be retrieved.

GUID

```
#define EFI_SIMPLE_POINTER_PROTOCOL_GUID \
  {0x31878c87,0xb75,0x11d5,0x9a,0x4f,0x0,0x90,0x27,0x3f,0xc1,0x4d}
```

Protocol Interface Structure

```
typedef struct _EFI_SIMPLE_POINTER_PROTOCOL {
    EFI_SIMPLE_POINTER_RESET        Reset;
    EFI_SIMPLE_POINTER_GET_STATE    GetState;
    EFI_EVENT                       WaitForInput;
    EFI_SIMPLE_INPUT_MODE           *Mode;
} EFI_SIMPLE_POINTER_PROTOCOL;
```

Parameters

Reset	Resets the pointer device. See the **Reset()** function description.
GetState	Retrieves the current state of the pointer device. See the **GetState()** function description.
WaitForInput	Event to use with **WaitForEvent()** to wait for input from the pointer device.
Mode	Pointer to **EFI_SIMPLE_POINTER_MODE** data. The type **EFI_SIMPLE_POINTER_MODE** is defined in "Related Definitions" below.

Related Definitions

```
//*******************************************************
// EFI_SIMPLE_POINTER_MODE
//*******************************************************
typedef struct {
    UINT64              ResolutionX;
    UINT64              ResolutionY;
    UINT64              ResolutionZ;
    BOOLEAN             LeftButton;
    BOOLEAN             RightButton;
} EFI_SIMPLE_POINTER_MODE;
```

The following data values in the **EFI_SIMPLE_POINTER_MODE** interface are read-only and are changed by using the appropriate interface functions:

ResolutionX	The resolution of the pointer device on the x-axis in counts/mm. If 0, then the pointer device does not support an x-axis.
ResolutionY	The resolution of the pointer device on the y-axis in counts/mm. If 0, then the pointer device does not support a y-axis.
ResolutionZ	The resolution of the pointer device on the z-axis in counts/mm. If 0, then the pointer device does not support a z-axis.
LeftButton	**TRUE** if a left button is present on the pointer device. Otherwise **FALSE**.
RightButton	**TRUE** if a right button is present on the pointer device. Otherwise **FALSE**.

Description

The **EFI_SIMPLE_POINTER_PROTOCOL** provides a set of services for a pointer device that can use used as an input device from an EFI application. The services include the ability to reset the pointer device, retrieve get the state of the pointer device, and retrieve the capabilities of the pointer device.

EFI_SIMPLE_POINTER.Reset()

Summary

Resets the pointer device hardware.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIMPLE_POINTER_RESET) (
  IN EFI_SIMPLE_POINTER_PROTOCOL  *This,
  IN BOOLEAN                      ExtendedVerification
  );
```

Parameters

This	A pointer to the **EFI_SIMPLE_POINTER_PROTOCOL** instance. Type **EFI_SIMPLE_POINTER_PROTOCOL** is defined in Section 10.10.
ExtendedVerification	Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description

This **Reset()** function resets the pointer device hardware.

As part of initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the *ExtendedVerification* flag is **TRUE** the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware and/or EFI driver to implement.

Status Codes Returned

EFI_SUCCESS	The device was reset.
EFI_DEVICE_ERROR	The device is not functioning correctly and could not be reset.

EFI_SIMPLE_POINTER.GetState()

Summary

Retrieves the current state of a pointer device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIMPLE_POINTER_GET_STATE)
  IN EFI_SIMPLE_POINTER_PROTOCOL    *This,
  IN OUT EFI_SIMPLE_POINTER_STATE   *State
  );
```

Parameters

This	A pointer to the **EFI_SIMPLE_POINTER_PROTOCOL** instance. Type **EFI_SIMPLE_POINTER_PROTOCOL** is defined in Section 10.10.
State	A pointer to the state information on the pointer device. Type **EFI_SIMPLE_POINTER_STATE** is defined in "Related Definitions" below.

Related Definitions

```
//*******************************************************
// EFI_SIMPLE_POINTER_STATE
//*******************************************************
typedef struct {
    INT32                RelativeMovementX;
    INT32                RelativeMovementY;
    INT32                RelativeMovementZ;
    BOOLEAN              LeftButton;
    BOOLEAN              RightButton;
} EFI_SIMPLE_POINTER_STATE;
```

RelativeMovementX	The signed distance in counts that the pointer device has been moved along the x-axis. The actual distance moved is *RelativeMovementX* / *ResolutionX* millimeters. If the *ResolutionX* field of the **EFI_SIMPLE_POINTER_MODE** structure is 0, then this pointer device does not support an x-axis, and this field must be ignored.

RelativeMovementY	The signed distance in counts that the pointer device has been moved along the y-axis. The actual distance moved is *RelativeMovementY* / *ResolutionY* millimeters. If the *ResolutionY* field of the **EFI_SIMPLE_POINTER_MODE** structure is 0, then this pointer device does not support a y-axis, and this field must be ignored.
RelativeMovementZ	The signed distance in counts that the pointer device has been moved along the z-axis. The actual distance moved is *RelativeMovementZ* / *ResolutionZ* millimeters. If the *ResolutionZ* field of the **EFI_SIMPLE_POINTER_MODE** structure is 0, then this pointer device does not support a z-axis, and this field must be ignored.
LeftButton	If **TRUE**, then the left button of the pointer device is being pressed. If **FALSE**, then the left button of the pointer device is not being pressed. If the *LeftButton* field of the **EFI_SIMPLE_POINTER_MODE** structure is **FALSE**, then this field is not valid, and must be ignored.
RightButton	If **TRUE**, then the right button of the pointer device is being pressed. If **FALSE**, then the right button of the pointer device is not being pressed. If the *RightButton* field of the **EFI_SIMPLE_POINTER_MODE** structure is **FALSE**, then this field is not valid, and must be ignored.

Description

The **GetState()** function retrieves the current state of a pointer device. This includes information on the buttons associated with the pointer device and the distance that each of the axes associated with the pointer device has been moved. If the state of the pointer device has not changed since the last call to **GetState()**, then **EFI_NOT_READY** is returned. If the state of the pointer device has changed since the last call to **GetState()**, then the state information is placed in *State*, and **EFI_SUCCESS** is returned. If a device error occurs while attempting to retrieve the state information, then **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	The state of the pointer device was returned in *State*.
EFI_NOT_READY	The state of the pointer device has not changed since the last call to **GetState()**.
EFI_DEVICE_ERROR	A device error occurred while attempting to retrieve the pointer device's current state.

10.11 EFI Simple Pointer Device Paths

An **EFI_SIMPLE_POINTER_PROTOCOL** must be installed on a handle for its services to be available to EFI Drivers and EFI Applications. In addition to the **EFI_SIMPLE_POINTER_PROTOCOL**, an **EFI_DEVICE_PATH** must also be installed on the same handle. See Chapter 5 of the EFI Specification for detailed description of the **EFI_DEVICE_PATH**.

A device path describes the location of a hardware component in a system from the processor's point of view. This includes the list of busses that lie between the processor and the pointer controller. The *EFI Specification* takes advantage of the *ACPI Specification* to name system components. The following set of examples shows sample device paths for a PS/2† mouse, a serial mouse, and a USB mouse.

Table 10-5 shows an example device path for a PS/2 mouse that is located behind a PCI to ISA bridge that is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. This device path consists of an ACPI Device Path Node for the PCI Root Bridge, a PCI Device Path Node for the PCI to ISA bridge, an ACPI Device Path Node for the PS/2 mouse, and a Device Path End Structure. The _HID and _UID of the first ACPI Device Path Node must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(7|0)/ACPI(PNP0F03,0)
```

Table 10-5. PS/2 Mouse Device Path

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x00	PCI Function
0x11	0x01	0x07	PCI Device

continued

Table 10-5. PS/2 Mouse Device Path (continued)

Byte Offset	Byte Length	Data	Description
0x12	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x13	0x01	0x01	Sub type – ACPI Device Path
0x14	0x02	0x0C	Length – 0x0C bytes
0x16	0x04	0x41D0, 0x0F03	_HID PNP0F03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x1A	0x04	0x0000	_UID
0x1E	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x1F	0x01	0xFF	Sub type – End of Entire Device Path
0x20	0x02	0x04	Length – 0x04 bytes

Table 10-6 shows an example device path for a serial mouse that is located on COM 1 behind a PCI to ISA bridge that is located at PCI device number 0x07 and PCI function 0x00. The PCI to ISA bridge is directly attached to a PCI root bridge, and the communications parameters for COM 1 are 1200 baud, no parity, 8 data bits, and 1 stop bit. This device path consists of an ACPI Device Path Node for the PCI Root Bridge, a PCI Device Path Node for the PCI to ISA bridge, an ACPI Device Path Node for COM 1, a UART Device Path Node for the communications parameters, an ACPI Device Path Node for the serial mouse, and a Device Path End Structure. The _HID and _UID of the first ACPI Device Path Node must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(7|0)/ACPI(PNP0501,0)/UART(1200N81)/ACPI(PNP0F01,0)
```

Table 10-6. Serial Mouse Device Path

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x08	0x04	0x0000	_UID

continued

Table 10-6. Serial Mouse Device Path (continued)

Byte Offset	Byte Length	Data	Description
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x00	PCI Function
0x11	0x01	0x07	PCI Device
0x12	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x13	0x01	0x01	Sub type – ACPI Device Path
0x14	0x02	0x0C	Length – 0x0C bytes
0x16	0x04	0x41D0, 0x0501	_HID PNP0501 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x1A	0x04	0x0000	_UID
0x1E	0x01	0x03	**Generic Device Path Header** – Messaging Device Path
0x1F	0x01	0x0E	Sub type – UART Device Path
0x20	0x02	0x13	Length – 0x13 bytes
0x22	0x04	0x00	Reserved
0x26	0x08	1200	Baud Rate
0x2E	0x01	0x08	Data Bits
0x2F	0x01	0x01	Parity
0x30	0x01	0x01	Stop Bits
0x31	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x32	0x01	0x01	Sub type – ACPI Device Path
0x33	0x02	0x0C	Length – 0x0C bytes
0x35	0x04	0x41D0, 0x0F01	_HID PNP0F01 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x39	0x04	0x0000	_UID
0x3D	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x3E	0x01	0xFF	Sub type – End of Entire Device Path
0x3F	0x02	0x04	Length – 0x04 bytes

Table 10-7 shows an example device path for a USB mouse that is behind a PCI to USB host controller that is located at PCI device number 0x07 and PCI function 0x02. The PCI to USB host controller is directly attached to a PCI root bridge. This device path consists of an ACPI Device Path Node for the PCI Root Bridge, a PCI Device Path Node for the PCI to USB controller, a USB Device Path Node, and a Device Path End Structure. The _HID and _UID of the first ACPI Device Path Node must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(7|2)/USB(0,0)
```

Table 10-7. USB Mouse Device Path

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x02	PCI Function
0x11	0x01	0x07	PCI Device
0x12	0x01	0x03	**Generic Device Path Header** – Type Messaging Device Path
0x13	0x01	0x05	Sub type – USB
0x14	0x02	0x06	Length – 0x06 bytes
0x16	0x01	0x00	USB Port Number
0x17	0x01	0x00	USB Endpoint Number
0x18	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x19	0x01	0xFF	Sub type – End of Entire Device Path
0x1A	0x02	0x04	Length – 0x04 bytes

10.12 Serial I/O Protocol

This section defines the Serial I/O protocol. This protocol is used to abstract byte stream devices.

SERIAL_IO_PROTOCOL

Summary

This protocol is used to communicate with any type of character-based I/O device.

GUID

```
#define SERIAL_IO_PROTOCOL \
     { BB25CF6F-F1D4-11D2-9A0C-0090273FC1FD }
```

Revision Number

```
#define SERIAL_IO_INTERFACE_REVISION  0x00010000
```

Protocol Interface Structure

```
typedef struct {
     UINT32                          Revision;
     EFI_SERIAL_RESET                Reset;
     EFI_SERIAL_SET_ATTRIBUTES       SetAttributes;
     EFI_SERIAL_SET_CONTROL_BITS     SetControl;
     EFI_SERIAL_GET_CONTROL_BITS     GetControl;
     EFI_SERIAL_WRITE                Write;
     EFI_SERIAL_READ                 Read;
     SERIAL_IO_MODE                  *Mode;
} SERIAL_IO_INTERFACE;
```

Parameters

Revision	The revision to which the **SERIAL_IO_INTERFACE** adheres. All future revisions must be backwards compatible. If a future version is not back wards compatible, it is not the same GUID.
Reset	Resets the hardware device.
SetAttributes	Sets communication parameters for a serial device. These include the baud rate, receive FIFO depth, transmit/receive time out, parity, data bits, and stop bit attributes.
SetControl	Sets the control bits on a serial device. These include Request to Send and Data Terminal Ready.
GetControl	Reads the status of the control bits on a serial device. These include Clear to Send, Data Set Ready, Ring Indicator, and Carrier Detect.
Write	Sends a buffer of characters to a serial device.
Read	Receives a buffer of characters from a serial device.

Mode	Pointer to **SERIAL_IO_MODE** data. Type **SERIAL_IO_MODE** is defined in "Related Definitions" below.

Related Definitions

```
//*******************************************************
// SERIAL_IO_MODE
//*******************************************************
typedef struct {
     UINT32                              ControlMask;

     // current Attributes
     UINT32                              Timeout;
     UINT64                              BaudRate;
     UINT32                              ReceiveFifoDepth;
     UINT32                              DataBits;
     UINT32                              Parity;
     UINT32                              StopBits;
} SERIAL_IO_MODE;
```

The data values in the **SERIAL_IO_MODE** are read-only and are updated by the code that produces the **SERIAL_IO_INTERFACE** protocol functions:

ControlMask	A mask of the Control bits that the device supports. The device must always support the Input Buffer Empty control bit.
Timeout	If applicable, the number of microseconds to wait before timing out a Read or Write operation.
BaudRate	If applicable, the current baud rate setting of the device; otherwise, baud rate has the value of zero to indicate that device runs at the device's designed speed.
ReceiveFifoDepth	The number of characters the device will buffer on input.
DataBits	The number of data bits in each character.
Parity	If applicable, this is the **EFI_PARITY_TYPE** that is computed or checked as each character is transmitted or received. If the device does not support parity the value is the default parity value.
StopBits	If applicable, the **EFI_STOP_BITS_TYPE** number of stop bits per character. If the device does not support stop bits the value is the default stop bit value.

```
//*******************************************************
// EFI_PARITY_TYPE
//*******************************************************
typedef enum {
     DefaultParity,
     NoParity,
     EvenParity,
     OddParity,
     MarkParity,
     SpaceParity
} EFI_PARITY_TYPE;

//*******************************************************
// EFI_STOP_BITS_TYPE
//*******************************************************
typedef enum {
     DefaultStopBits,
     OneStopBit,            // 1 stop bit
     OneFiveStopBits,       // 1.5 stop bits
     TwoStopBits            // 2 stop bits
} EFI_STOP_BITS_TYPE;
```

Description

The Serial I/O protocol is used to communicate with UART-style serial devices. These can be standard UART serial ports in PC-AT systems, serial ports attached to a USB interface, or potentially any character-based I/O device.

The Serial I/O protocol can control byte I/O style devices from a generic device to a device with features such as a UART. As such many of the serial I/O features are optional to allow for the case of devices that do not have UART controls. Each of these options is called out in the specific serial I/O functions.

The default attributes for all UART-style serial device interfaces are: 115,200 baud, a 1 byte receive FIFO, a 1,000,000 microsecond timeout per character, no parity, 8 data bits, and 1 stop bit. Flow control is the responsibility of the software that uses the protocol. Hardware flow control can be implemented through the use of the `GetControl()` and `SetControl()` functions (described below) to monitor and assert the flow control signals. The XON/XOFF flow control algorithm can be implemented in software by inserting XON and XOFF characters into the serial data stream as required.

Special care must be taken if a significant amount of data is going to be read from a serial device. Since EFI drivers are polled mode drivers, characters received on a serial device might be missed. It is the responsibility of the software that uses the protocol to check for new data often enough to guarantee that no characters will be missed. The required polling frequency depends on the baud rate of the connection and the depth of the receive FIFO.

SERIAL_IO.Reset()

Summary

Resets the serial device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SERIAL_RESET) (
      IN SERIAL_IO_INTERFACE      *This
      );
```

Parameters

This	A pointer to the **SERIAL_IO_INTERFACE** instance. Type **SERIAL_IO_INTERFACE** is defined in Section 10.12.

Description

The **Reset()** function resets the hardware of a serial device.

Status Codes Returned

EFI_SUCCESS	The serial device was reset.
EFI_DEVICE_ERROR	The serial device could not be reset.

SERIAL_IO.SetAttributes()

Summary

Sets the baud rate, receive FIFO depth, transmit/receive time out, parity, data bits, and stop bits on a serial device.

```
EFI_STATUS
(EFIAPI *EFI_SERIAL_SET_ATTRIBUTES) (
     IN SERIAL_IO_INTERFACE      *This,
     IN UINT64                   BaudRate,
     IN UINT32                   ReceiveFifoDepth,
     IN UINT32                   Timeout
     IN EFI_PARITY_TYPE          Parity,
     IN UINT8                    DataBits,
     IN EFI_STOP_BITS_TYPE       StopBits
     );
```

Parameters

This	A pointer to the **SERIAL_IO_INTERFACE** instance. Type **SERIAL_IO_INTERFACE** is defined in Section 10.12.
BaudRate	The requested baud rate. A *BaudRate* value of 0 will use the device's default interface speed.
ReceiveFifoDepth	The requested depth of the FIFO on the receive side of the serial interface. A *ReceiveFifoDepth* value of 0 will use the device's default FIFO depth.
Timeout	The requested time out for a single character in microseconds. This timeout applies to both the transmit and receive side of the interface. A *Timeout* value of 0 will use the device's default time out value.
Parity	The type of parity to use on this serial device. A *Parity* value of **DefaultParity** will use the device's default parity value. Type **EFI_PARITY_TYPE** is defined in "Related Definitions" in Section 10.12.
DataBits	The number of data bits to use on this serial device. A *DataBits* value of 0 will use the device's default data bit setting.
StopBits	The number of stop bits to use on this serial device. A *StopBits* value of **DefaultStopBits** will use the device's default number of stop bits. Type **EFI_STOP_BITS_TYPE** is defined in "Related Definitions" in Section 10.12.

Description

The **`SetAttributes()`** function sets the baud rate, receive-FIFO depth, transmit/receive time out, parity, data bits, and stop bits on a serial device.

The controller for a serial device is programmed with the specified attributes. If the *`Parity`*, *`DataBits`*, or *`StopBits`* values are not valid, then an error will be returned. If the specified *`BaudRate`* is below the minimum baud rate supported by the serial device, an error will be returned. The nearest baud rate supported by the serial device will be selected without exceeding the *`BaudRate`* parameter. If the specified *`ReceiveFifoDepth`* is below the smallest FIFO size supported by the serial device, an error will be returned. The nearest FIFO size supported by the serial device will be selected without exceeding the *`ReceiveFifoDepth`* parameter.

Status Codes Returned

EFI_SUCCESS	The new attributes were set on the serial device.
EFI_INVALID_PARAMETER	One or more of the attributes has an unsupported value.
EFI_DEVICE_ERROR	The serial device is not functioning correctly.

SERIAL_IO.SetControl()

Summary

Sets the control bits on a serial device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SERIAL_SET_CONTROL) (
     IN SERIAL_IO_INTERFACE     *This,
     IN UINT32                  Control
     );
```

Parameters

This	A pointer to the **SERIAL_IO_INTERFACE** instance. Type **SERIAL_IO_INTERFACE** is defined in Section 10.12.
Control	Sets the bits of *Control* that are settable. See "Related Definitions" below.

Related Definitions

```
//*******************************************************
// CONTROL BITS
//*******************************************************

#define EFI_SERIAL_CLEAR_TO_SEND                  0x0010
#define EFI_SERIAL_DATA_SET_READY                 0x0020
#define EFI_SERIAL_RING_INDICATE                  0x0040
#define EFI_SERIAL_CARRIER_DETECT                 0x0080
#define EFI_SERIAL_REQUEST_TO_SEND                0x0002
#define EFI_SERIAL_DATA_TERMINAL_READY            0x0001
#define EFI_SERIAL_INPUT_BUFFER_EMPTY             0x0100
#define EFI_SERIAL_OUTPUT_BUFFER_EMPTY            0x0200
#define EFI_SERIAL_HARDWARE_LOOPBACK_ENABLE       0x1000
#define EFI_SERIAL_SOFTWARE_LOOPBACK_ENABLE       0x2000
#define EFI_SERIAL_HARDWARE_FLOW_CONTROL_ENABLE   0x4000
```

Description

The **SetControl()** function is used to assert or deassert the control signals on a serial device. The following signals are set according their bit settings:

- Request to Send
- Data Terminal Ready

Only the **REQUEST_TO_SEND**, **DATA_TERMINAL_READY**, **HARDWARE_LOOPBACK_ENABLE**, **SOFTWARE_LOOPBACK_ENABLE**, and **HARDWARE_FLOW_CONTROL_ENABLE** bits can be set with **SetControl()**. All the bits can be read with **GetControl()**.

Status Codes Returned

EFI_SUCCESS	The new control bits were set on the serial device.
EFI_UNSUPPORTED	The serial device does not support this operation.
EFI_DEVICE_ERROR	The serial device is not functioning correctly.

SERIAL_IO.GetControl()

Summary

Retrieves the status of the control bits on a serial device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SERIAL_GET_CONTROL) (
     IN SERIAL_IO_INTERFACE      *This,
     OUT UINT32                  *Control
     );
```

Parameters

This	A pointer to the **SERIAL_IO_INTERFACE** instance. Type **SERIAL_IO_INTERFACE** is defined in Section 10.12.
Control	A pointer to return the current control signals from the serial device. See "Related Definitions" below.

Related Definitions

```
//*******************************************************
// CONTROL BITS
//*******************************************************

#define EFI_SERIAL_CLEAR_TO_SEND                 0x0010
#define EFI_SERIAL_DATA_SET_READY                0x0020
#define EFI_SERIAL_RING_INDICATE                 0x0040
#define EFI_SERIAL_CARRIER_DETECT                0x0080
#define EFI_SERIAL_REQUEST_TO_SEND               0x0002
#define EFI_SERIAL_DATA_TERMINAL_READY           0x0001
#define EFI_SERIAL_INPUT_BUFFER_EMPTY            0x0100
#define EFI_SERIAL_OUTPUT_BUFFER_EMPTY           0x0200
#define EFI_SERIAL_HARDWARE_LOOPBACK_ENABLE      0x1000
#define EFI_SERIAL_SOFTWARE_LOOPBACK_ENABLE      0x2000
#define EFI_SERIAL_HARDWARE_FLOW_CONTROL_ENABLE  0x4000
```

Description

The **GetControl()** function retrieves the status of the control bits on a serial device.

Status Codes Returned

EFI_SUCCESS	The control bits were read from the serial device.
EFI_DEVICE_ERROR	The serial device is not functioning correctly.

SERIAL_IO.Write()

Summary

Writes data to a serial device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SERIAL_WRITE) (
      IN SERIAL_IO_INTERFACE      *This,
      IN OUT UINTN                *BufferSize,
      IN VOID                     *Buffer
      );
```

Parameters

This	A pointer to the **SERIAL_IO_INTERFACE** instance. Type **SERIAL_IO_INTERFACE** is defined in Section 10.12.
BufferSize	On input, the size of the *Buffer*. On output, the amount of data actually written.
Buffer	The buffer of data to write.

Description

The **Write()** function writes the specified number of bytes to a serial device. If a time out error occurs while data is being sent to the serial port, transmission of this buffer will terminate, and **EFI_TIMEOUT** will be returned. In all cases the number of bytes actually written to the serial device is returned in *BufferSize*.

Status Codes Returned

EFI_SUCCESS	The data was written.
EFI_DEVICE_ERROR	The device reported an error.
EFI_TIMEOUT	The data write was stopped due to a timeout.

SERIAL_IO.Read()

Summary

Reads data from a serial device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SERIAL_READ) (
      IN SERIAL_IO_INTERFACE       *This,
      IN OUT UINTN                 *BufferSize,
      OUT VOID                     *Buffer
      );
```

Parameters

This	A pointer to the **SERIAL_IO_INTERFACE** instance. Type **SERIAL_IO_INTERFACE** is defined in Section 10.12.
BufferSize	On input, the size of the *Buffer*. On output, the amount of data returned in *Buffer*.
Buffer	The buffer to return the data into.

Description

The **Read()** function reads a specified number of bytes from a serial device. If a time out error or an overrun error is detected while data is being read from the serial device, then no more characters will be read, and an error will be returned. In all cases the number of bytes actually read is returned in *BufferSize*.

Status Codes Returned

EFI_SUCCESS	The data was read.
EFI_DEVICE_ERROR	The serial device reported an error.
EFI_TIMEOUT	The operation was stopped due to a timeout or overrun.

11
Protocols - Bootable Image Support

11.1 LOAD_FILE Protocol

This section defines the Load File protocol. This protocol is designed to allow code running in the EFI boot services environment to find and load other modules of code.

LOAD_FILE Protocol

Summary

Is used to obtain files from arbitrary devices.

GUID

```
#define LOAD_FILE_PROTOCOL \
      {56EC3091-954C-11d2-8E3F-00A0C969723B}
```

Protocol Interface Structure

```
typedef struct {
    EFI_LOAD_FILE                    LoadFile;
} EFI_LOAD_FILE_INTERFACE;
```

Parameters

LoadFile	Causes the driver to load the requested file. See the **LoadFile()** function description.

Description

The **EFI_LOAD_FILE** protocol is a simple protocol used to obtain files from arbitrary devices.

When the firmware is attempting to load a file, it first attempts to use the device's Simple File System protocol to read the file. If the file system protocol is found, the firmware implements the policy of interpreting the File Path value of the file being loaded. If the device does not support the file system protocol, the firmware then attempts to read the file via the **EFI_LOAD_FILE** protocol and the **LoadFile()** function. In this case the **LoadFile()** function implements the policy of interpreting the File Path value.

LOAD_FILE.LoadFile()

Summary

Causes the driver to load a specified file.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_LOAD_FILE) (
     IN EFI_LOAD_FILE_INTERFACE  *This,
     IN EFI_DEVICE_PATH          *FilePath,
     IN BOOLEAN                  BootPolicy,
     IN OUT UINTN                *BufferSize,
     IN VOID                     *Buffer          OPTIONAL
     );
```

Parameters

This	Indicates a pointer to the calling context. Type **EFI_LOAD_FILE_INTERFACE** is defined in Section 11.1.
FilePath	The device specific path of the file to load. Type **EFI_DEVICE_PATH** is defined in Chapter 8.
BootPolicy	If **TRUE**, indicates that the request originates from the boot manager, and that the boot manager is attempting to load *FilePath* as a boot selection. If **FALSE**, then *FilePath* must match an exact file to be loaded.
BufferSize	On input the size of *Buffer* in bytes. On output with a return code of **EFI_SUCCESS**, the amount of data transferred to *Buffer*. On output with a return code of **EFI_BUFFER_TOO_SMALL**, the size of *Buffer* required to retrieve the requested file.
Buffer	The memory buffer to transfer the file to. If *Buffer* is **NULL**, then no the size of the requested file is returned in *BufferSize*.

Description

The **LoadFile()** function interprets the device-specific *FilePath* parameter, returns the entire file into *Buffer*, and sets *BufferSize* to the amount of data returned. If *Buffer* is **NULL**, then the size of the file is returned in *BufferSize*. If *Buffer* is not **NULL**, and *BufferSize* is not large enough to hold the entire file, then **EFI_BUFFER_TOO_SMALL** is returned, and *BufferSize* is updated to indicate the size of the buffer needed to obtain the file. In this case, no data is returned in *Buffer*.

If *BootPolicy* is **FALSE** the *FilePath* must match an exact file to be loaded. If no such file exists, **EFI_NOT_FOUND** is returned. If *BootPolicy* is **FALSE**, and an attempt is being made to perform a network boot through the PXE Base Code protocol, **EFI_UNSUPPORTED** is returned.

If *BootPolicy* is **TRUE** the firmware's boot manager is attempting to load an EFI image that is a boot selection. In this case, *FilePath* contains the file path value in the boot selection option. Normally the firmware would implement the policy on how to handle an inexact boot file path; however, since in this case the firmware cannot interpret the file path, the **LoadFile()** function is responsible for implementing the policy. For example, in the case of a network boot through the PXE Base Code protocol, *FilePath* merely points to the root of the device, and the firmware interprets this as wanting to boot from the first valid loader. The following is list of events that **LoadFile()** will implement for a PXE boot:

- Perform DHCP.
- Optionally prompt the user with a menu of boot selections.
- Discover the boot server and the boot file.
- Download the boot file into *Buffer* and update *BufferSize* with the size of the boot file.

Status Codes Returned

EFI_SUCCESS	The file was loaded.
EFI_UNSUPPORTED	The device does not support the provided *BootPolicy*.
EFI_INVALID_PARAMETER	*FilePath* is not a valid device path, or *BufferSize* is **NULL**.
EFI_NO_SUCH_MEDIA	No medium was present to load the file.
EFI_DEVICE_ERROR	The file was not loaded due to a device error.
EFI_NO_RESPONSE	The remote system did not respond.
EFI_NOT_FOUND	The file was not found.
EFI_ABORTED	The file load process was manually cancelled.

11.2 File System Format

The file system supported by the Extensible Firmware Interface is based on the FAT file system. EFI defines a specific version of FAT that is explicitly documented and testable. Conformance to the EFI specification and its associate reference documents is the only definition of FAT that needs to be implemented to support EFI. To differentiate the EFI file system from pure FAT, a new partition file system type has been defined.

EFI encompasses the use of FAT32 for a system partition, and FAT12 or FAT16 for removable media. The FAT32 system partition is identified by an OS type value other than that used to identify previous versions of FAT. This unique partition type distinguishes an EFI defined file system from a normal FAT file system. The file system supported by EFI includes support for long file names.

The definition of the EFI file system will be maintained by specification and will not evolve over time to deal with errata or variant interpretations in OS file system drivers or file system utilities. Future enhancements and compatibility enhancements to FAT will not be automatically included in EFI file systems. The EFI file system is a target that is fixed by the EFI specification, and other specifications explicitly referenced by the EFI specification.

For more information about the EFI file system and file image format, visit the web site from which this document was obtained.

11.2.1 System Partition

A System Partition is a partition in the conventional sense of a partition on a legacy Intel architecture system. For a hard disk, a partition is a contiguous grouping of sectors on the disk where the starting sector and size are defined by the Master Boot Record (MBR), which resides on the first sector of the hard disk. For a diskette (floppy) drive, a partition is defined to be the entire media. A System Partition can reside on any media that is supported by EFI Boot Services.

A System Partition supports backward compatibility with legacy Intel architecture systems by reserving the first block (sector) of the partition for compatibility code. On legacy Intel architecture systems, the first block (sector) of a partition is loaded into memory and execution is transferred to this code. EFI firmware does not execute the code in the MBR. The EFI firmware contains knowledge about the partition structure of various devices, and can understand legacy MBR, EFI partition record, and “El Torito.”

The System Partition contains directories, data files, and EFI Images. EFI Images can contain an EFI OS Loader, an EFI Driver to extend platform firmware capability, or an EFI Application that provides a transient service to the system. EFI Applications could include things such as a utility to create partitions or extended diagnostics. A System Partition can also support data files, such as error logs, that can be defined and used by various OS or system firmware software components.

11.2.1.1 File System Format

The first block (sector) of a partition contains a data structure called the BIOS Parameter Block, BPB, that defines the type and location of FAT file system on the drive. The BPB contains a data structure that defines the size of the media, the size of reserved space, the number of FAT tables, and the location and size of the root directory (not used in FAT32). The first block (sector) also contains code that will be executed as part of the boot process on a legacy Intel architecture system. This code in the first block (sector) usually contains code that can read a file from the root directory into memory and transfer control to it. Since EFI firmware contains a file system driver, EFI firmware can load any file from the file system with out needing to execute any code from the media.

The EFI firmware must support the FAT32, FAT16, and FAT12 variants of the EFI file system. What variant of EFI FAT to use is defined by the size of the media. The rules defining the relationship between media size and FAT variants is defined in the specification for the EFI file system.

11.2.1.2 File Names

FAT stores file names in two formats. The original FAT format limited file names to eight characters with three extension characters. This type of file name is called an 8.3, pronounced eight dot three, file name. FAT was extended to include support for long file names (LFN).

FAT 8.3 file names are always stored as uppercase ASCII characters. LFN can either be stored as ASCII or Unicode and are stored case sensitive. The string that was used to open or create the file is stored directly into LFN. FAT defines that all files in a directory must have a unique name, and unique is defined as a case insensitive match. The following are examples of names that are considered to be the same and cannot exist in a single directory:

- “ThisIsAnExampleDirectory.Dir”
- “thisisanexamppledirectory.dir”
- THISISANEXAMPLEDIRECTORY.DIR
- ThisIsAnExampleDirectory.DIR

11.2.1.3 Directory Structure

An EFI system partition that is present on a hard disk must contain an EFI defined directory in the root directory. This directory is named **`EFI`**. All OS loaders and applications will be stored in subdirectories below **`EFI`**. Applications that are loaded by other applications or drivers are not required to be stored in any specific location in the EFI system partition. The choice of the subdirectory name is up to the vendor, but all vendors must pick names that do not collide with any other vendor’s subdirectory name. This applies to system manufacturers, operating system vendors, BIOS vendors, and third party tool vendors, or any other vendor that wishes to install files on an EFI system partition. There must also only be one executable EFI image for each supported processor architecture in each vendor subdirectory. This guarantees that there is only one image that can be loaded from a vendor subdirectory by the EFI Boot Manager. If more than one executable EFI image is present, then the boot behavior for the system will not be deterministic. There may also be an optional vendor subdirectory called **`BOOT`**.

This directory contains EFI images that aide in recovery if the boot selections for the software installed on the EFI system partition are ever lost. Any additional EFI executables must be in subdirectories below the vendor subdirectory. The following is a sample directory structure for an EFI system partition present on a hard disk.

```
\EFI
     \<OS Vendor 1 Directory>
          <OS Loader Image>
     \<OS Vendor 2 Directory>
          <OS Loader Image>
     . . .
     \<OS Vendor N Directory>
          <OS Loader Image>
     \<OEM Directory>
          <OEM Application Image>
     \<BIOS Vendor Directory>
          <BIOS Vendor Application Image>
     \<Third Party Tool Vendor Directory>
          <Third Party Tool Vendor Application Image>
     \BOOT
          BOOT{machine type short name}.EFI
```

For removable media devices there must be only one EFI system partition, and that partition must contain an EFI defined directory in the root directory. The directory will be named **EFI**. All OS loaders and applications will be stored in a subdirectory below **EFI** called **BOOT**. There must only be one executable EFI image for each supported processor architecture in the **BOOT** directory. For removable media to be bootable under EFI, it must be built in accordance with the rules laid out in Section 17.4.1.1. This guarantees that there is only one image that can be automatically loaded from a removable media device by the EFI Boot Manager. Any additional EFI executables must be in directories other than **BOOT**. The following is a sample directory structure for an EFI system partition present on a removable media device.

```
\EFI
     \BOOT
          BOOT{machine type short name}.EFI
```

11.2.2 Partition Discovery

EFI requires the firmware to be able to parse legacy master boot records, the new GUID Partition Table (GPT), and El Torito logical device volumes. The EFI firmware produces a logical BLOCK_IO device for each EFI Partition Entry, El Torito logical device volume, and if no EFI Partition Table is present any partitions found in the partition tables. Logical block address zero of the BLOCK_IO device will correspond to the first logical block of the partition. See Figure 11-1.

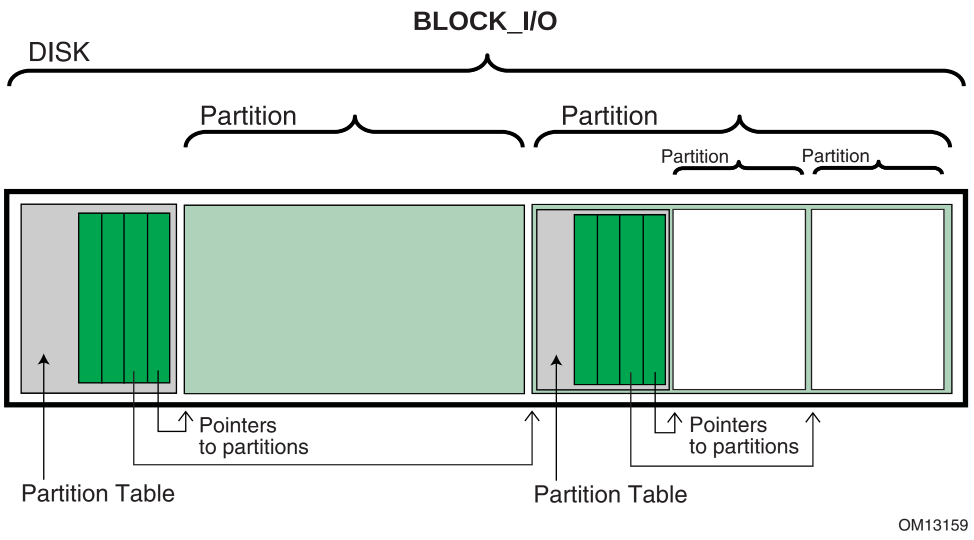


Figure 11-1. Nesting of Legacy MBR Partition Records

The following is the order in which a block device must be scanned to determine if it contains partitions. When a check for a valid partitioning scheme succeeds, the search terminates.

1. Check for GUID Partition Table Headers.
2. Follow ISO-9660 specification to search for ISO-9660 volume structures on the magic LBA.
   — Check for an “El Torito” volume extension and follow the “El Torito” CD-ROM specification.
3. If none of the above, check LBA 0 for a legacy MBR partition table.
4. No partition found on device.

EFI supports the nesting of legacy MBR partitions, by allowing any legacy MBR partition to contain more legacy MBR partitions. This is accomplished by supporting the same partition discovery algorithm on every logical block device. It should be noted that the GUID Partition Table does not allow nesting of GUID Partition Table Headers. Nesting is not needed since a GUID Partition Table Header can support an arbitrary number of partitions (the addressability limits of a 64-bit LBA is the limiting factor).

11.2.2.1 EFI Partition Header

EFI defines a new partitioning scheme that must be supported by EFI firmware. The following list outlines the advantages of using the GUID Partition Table over the legacy MBR partition table:

- Logical Block Addressing is 64 bits.
- Supports many partitions.
- Uses a primary and backup table for redundancy.
- Uses version number and size fields for future expansion.
- Uses CRC32 fields for improved data integrity.
- Defines a GUID for uniquely identifying each partition.
- Uses a GUID and attributes to define partition content type.
- Each partition contains a 36 Unicode character human readable name.

The EFI partitioning scheme is depicted in Figure 11-2. The GUID Partition Table Header (see Table 11-1) starts with a signature and a revision number that specifies which version of the EFI specification defines the data bytes in the partition header. The GUID Partition Table Header contains a header size field that is used in calculating the CRC32 that confirms the integrity of the GUID Partition Table Header. While the GUID Partition Table Header's size may increase in the future it cannot span more than one block on the device.

Two GUID Partition Table Header structures are stored on the device: the primary and the backup. The primary GUID Partition Table Header must be located in block 1 of the logical device, and the backup GUID Partition Table Header must be located in the last block of the logical device. Within the GUID Partition Table Header there are the *MyLBA* and *AlternateLBA* fields. The *MyLBA* field contains the logical block address of the GUID Partition Table Header itself, and the *AlternateLBA* field contains the logical block address of the other GUID Partition Table Header. For example, the primary GUID Partition Table Header's *MyLBA* value would be 1 and its *AlternateLBA* would be the value for the last block of the logical device. The backup GUID Partition Table Header's fields would be reversed.

The GUID Partition Table Header defines the range of logical block addresses that are usable by Partition Entries. This range is defined to be inclusive of *FirstUsableLBA* through *LastUsableLBA* on the logical device. All data stored on the volume must be stored between the *FirstUsableLBA* through *LastUsableLBA*, and only the data structures defined by EFI to manage partitions may reside outside of the usable space. The value of *DiskGUID* is a GUID that uniquely identifies the entire GUID Partition Table Header and all its associated storage. This value can be used to uniquely identify the disk. The start of the GUID Partition Entry array is located at the logical block address *PartitionEntryLBA*. The size of a GUID Partition Entry element is defined in the GUID Partition Table Header. There is a 32-bit CRC of the GUID Partition Entry array that is stored in the GUID Partition Table Header in *PartitionEntryArrayCRC*. The size of the GUID Partition Entry array is the *PartitionEntrySize* multiplied by *NumberOfPartitionEntries*. When a GUID Partition Entry is updated, the *PartitionEntryArrayCRC* must be updated. When the *PartitionEntryArrayCRC* is updated, the GUID Partition Table Header CRC must also be updated, since the *PartitionEntryArrayCRC* is stored in the GUID Partition Table Header.

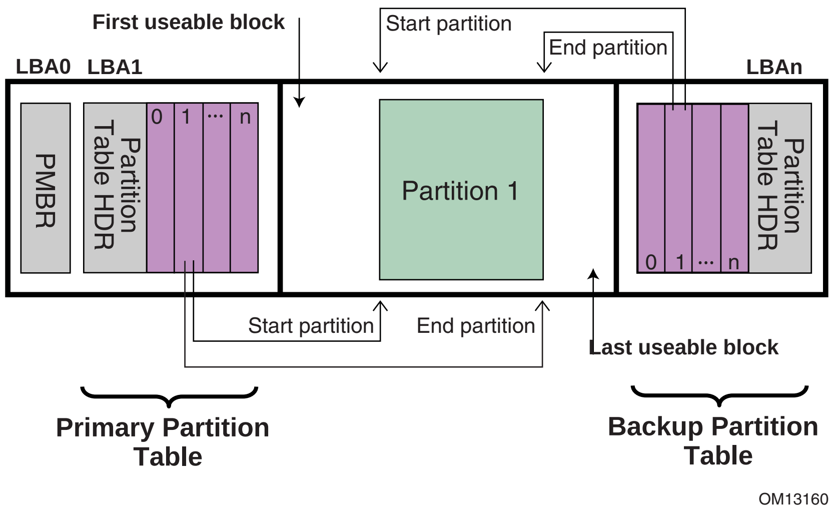


Figure 11-2. GUID Partition Table (GPT) Scheme

The primary GUID Partition Entry array must be located after the primary GUID Partition Table Header and end before the *FirstUsableLBA*. The backup GUID Partition Entry array must be located after the *LastUsableLBA* and end before the backup GUID Partition Table Header. Therefore the primary and backup GUID Partition Entry arrays are stored in separate locations on the disk. GUID Partition Entries define a partition that is contained in a range that is within the usable space declared by the GUID Partition Table Header. Zero or more GUID Partition Entries may be in use in the GUID Partition Entry array. Each defined partition must not overlap with any other defined partition. If all the fields of a GUID Partition Entry are zero, the entry is not in use. A minimum of 16,384 bytes of space must be reserved for the GUID Partition Entry array. Typically the first useable block will start at an LBA greater than or equal to 34, assuming the LBA block size is 512 bytes.

Table 11-1. GUID Partition Table Header

Mnemonic	Byte Offset	Byte Length	Description
Signature	0	8	Identifies EFI-compatible partition table header. This value must contain the string “EFI PART,” 0x5452415020494645.
Revision	8	4	The specification revision number that this header complies to. For version 1.0 of the specification the correct value is 0x00010000.
HeaderSize	12	4	Size in bytes of the GUID Partition Table Header.
HeaderCRC32	16	4	CRC32 checksum for the GUID Partition Table Header structure. The range defined by *HeaderSize* is “check-summed.”
Reserved	20	4	Must be zero.

continued

Table 11-1. GUID Partition Table Header (continued)

Mnemonic	Byte Offset	Byte Length	Description
MyLBA	24	8	The LBA that contains this data structure.
AlternateLBA	32	8	LBA address of the alternate GUID Partition Table Header.
FirstUsableLBA	40	8	The first usable logical block that may be contained in a GUID Partition Entry.
LastUsableLBA	48	8	The last usable logical block that may be contained in a GUID Partition Entry.
DiskGUID	56	16	GUID that can be used to uniquely identify the disk.
PartitionEntryLBA	72	8	The starting LBA of the GUID Partition Entry array.
NumberOfPartitionEntries	80	4	The number of Partition Entries in the GUID Partition Entry array.
SizeOfPartitionEntry	84	4	The size, in bytes, of each the GUID Partition Entry structures in the GUID Partition Entry array. Must be a multiple of 8.
PartitionEntryArrayCRC32	88	4	The CRC32 of the GUID Partition Entry array. Starts at Partition Entry LBA and is `NumberOfPartitionEntries * SizeOfPartitionEntry` in byte length.
Reserved	92	BlockSize – 92	The rest of the block is reserved by EFI and must be zero.

The following test must be performed to determine if a GUID Partition Table is valid:

- Check the GUID Partition Table Signature
- Check the GUID Partition Table CRC
- Check that the *MyLBA* entry points to the LBA that contains the GUID Partition Table
- Check the CRC of the GUID Partition Entry Array

If the GUID Partition Table is the primary table, stored at LBA 1:

- Check the *AlternateLBA* to see if it is a valid GUID Partition Table

If the primary GUID Partition Table is corrupt:

- Check the last LBA of the device to see if it has a valid GUID Partition Table.
- If valid backup GUID Partition Table found, restore primary GUID Partition Table.

Any software that updates the primary GUID Partition Table Header must also update the backup GUID Partition Table Header. The order of the update of the GUID Partition Table Header and its associated GUID Partition Entry array is not important, since all the CRCs are stored in the GUID Partition Table Header. However, the primary GUID Partition Table Header and GUID Partition Entry array must always be updated before the backup.

If the primary GUID Partition Table is invalid the backup GUID Partition Table is located on the last logical block on the disk. If the backup GUID Partition Table is valid it must be used to restore the primary GUID Partition Table. If the primary GUID Partition Table is valid and the backup GUID Partition Table is invalid software must restore the backup GUID Partition Table. If both the primary and backup GUID Partition Table is corrupted this block device is defined as not having a valid GUID Partition Header.

The primary and backup GUID Partition Tables must be valid before an attempt is made to grow the size of a physical volume. This is due to the GUID Partition Table recovery scheme depending on locating the backup GUID Partition Table at the end of the physical device. A volume may grow in size when disks are added to a RAID device. As soon as the volume size is increased the backup GUID Partition Table must be moved to the end of the volume and the primary and backup GUID Partition Table Headers must be updated to reflect the new volume size.

Table 11-2. GUID Partition Entry

Mnemonic	Byte Offset	Byte Length	Description
Partition Type Guid	0	16	Unique ID that defines the purpose and type of this Partition. A value of zero defines that this partition record is not being used.
Unique Partition Guid	16	16	GUID that is unique for every partition record. Every partition ever created will have a unique GUID. This GUID must be assigned when the GUID Partition Entry is created. The GUID Partition Entry is created when ever the *NumberOfPartitionEntries* in the GUID Partition Table Header is increased to include a larger range of addresses.
StartingLBA	32	8	Starting LBA of the partition defined by this record.
EndingLBA	40	8	Ending LBA of the partition defined by this record.
Attributes	48	8	Attribute bits, all bits reserved by EFI.
Partition Name	56	72	Unicode string.

The *SizeOfPartitionEntry* variable in the GUID Partition Table Header defines the size of a GUID Partition Entry. The GUID Partition Entry starts in the first byte of the GUID Partition Entry and any unused space at the end of the defined partition entry is reserved space and must be set to zero.

Each partition record contains a Unique Partition GUID variable that uniquely identifies every partition that will ever be created. Any time a new partition record is created a new GUID must be generated for that partition, and every partition is guaranteed to have a unique GUID. The partition record also contains 64-bit logical block addresses for the starting and ending block of the partition. The partition is defined as all the logical blocks inclusive of the starting and ending usable LBA defined in the GUID Partition Table Header. The partition record contains a partition type GUID that identifies the contents of the partition. This GUID is similar to the OS type field in the legacy MBR. Each file system must publish its unique GUID. The partition record also contains Attributes that can be used by utilities to make broad inferences about the usage of a partition. A

36-character Unicode string is also included, so that a human readable string can be used to represent what information is stored on the partition. This allows third party utilities to give human readable names to partitions.

The firmware must add the *PartitionTypeGuid* to the handle of every active GPT partition using **InstallProtocolInterface()**. This will allow drivers and applications, including OS loaders, to easily search for handles that represent EFI System Partitions or vendor specific partition types.

A utility that makes a binary copy of a disk that is formatted with GPT must generate a new *DiskGUID* in the Partition Table Headers. In addition, new *UniquePartitionGuids* must be generated for each GUID Partition Entry.

Table 11-3. Defined GUID Partition Entry - Partition Type GUIDs

Description	GUID Value
Unused Entry	00000000-0000-0000-0000-000000000000
EFI System Partition	C12A7328-F81F-11d2-BA4B-00A0C93EC93B
Partition containing a legacy MBR	024DEE41-33E7-11d3-9D69-0008C781F39F

OS vendors need to generate their own GUIDs to identify their partition types.

Table 11-4. Defined GUID Partition Entry - Attributes

Bits	Description
Bit 0	Required for the platform to function. The system cannot function normally if this partition is removed. This partition should be considered as part of the hardware of the system, and if it is removed the system may not boot. It may contain diagnostics, recovery tools, or other code or data that is critical to the functioning of a system independent of any OS.
Bits1-47	Undefined and must be zero. Reserved for expansion by future versions of the EFI specification.
Bits 48-63	Reserved for GUID specific use. The use of these bits will vary depending on the *PartitionTypeGuid*. Only the owner of the *PartitionTypeGuid* is allowed to modify these bits. They must be preserved if Bits 0–47 are modified.

11.2.2.2 ISO-9660 and El Torito

IS0-9660 is the industry standard low level format used on CD-ROM and DVD-ROM. CD-ROM format is completely described by the “El Torito” Bootable CD-ROM Format Specification Version 1.0. To boot from a CD-ROM or DVD-ROM in the boot services environment, an EFI System partition is stored in a “no emulation” mode as defined by the “El Torito” specification. A Platform ID of 0xEF hex indicates an EFI System Partition. The Platform ID is in either the Section Header Entry or the Validation Entry of the Booting Catalog as defined by the “El Torito” specification. EFI differs from “El Torito” “no emulation” mode in that it does not load the “no emulation” image into memory and jump to it. EFI interprets the “no emulation” image as an EFI system partition. EFI interprets the Sector Count in the Initial/Default Entry or the Section Header Entry to be the size of the EFI system partition. If the value of Sector Count is set to 0 or 1, EFI will assume the system partition consumes the space from the beginning of the “no emulation” image to the end of the CD-ROM.

DVD-ROM images formatted as required by the UDF 2.00 specification (*OSTA Universal Disk Format Specification,* Revision 2.00) can be booted by EFI. EFI supports booting from an ISO-9660 file system that conforms to the *"El Torito" Bootable CD-ROM Format Specification* on a DVD-ROM. A DVD-ROM that contains an ISO-9660 file system is defined as a "UDF Bridge" disk. Booting from CD-ROM and DVD-ROM is accomplished using the same methods.

Since the EFI file system definition does not use the same Initial/Default entry as a legacy CD-ROM it is possible to boot Intel architecture personal computers using an EFI CD-ROM or DVD-ROM. The inclusion of boot code for Intel architecture personal computers is optional and not required by EFI.

11.2.2.3 Legacy Master Boot Record

The legacy master boot record is the first block (sector) on the disk media. The boot code on the MBR is not executed by EFI firmware. The MBR may optionally contain a signature located as defined in Table 11-5. The MBR signature must be maintained by operating systems, and is never maintained by EFI firmware. The unique signature in the MBR is only 4 bytes in length, so it is not a GUID. EFI does not specify the algorithm that is used to generate the unique signature. The uniqueness of the signature is defined as all disks in a given system having a unique value in this field.

Table 11-5. Legacy Master Boot Record

Mnemonic	Byte Offset	Byte Length	Description
BootCode	0	440	Code used on legacy Intel architecture system to select a partition record and load the first block (sector) of the partition pointed to by the partition record. This code is not executed on EFI systems.
UniqueMBRSignature	440	4	Unique Disk Signature, this is an optional feature and not on all hard drives. This value is always written by the OS and is never written by EFI firmware.
Unknown	444	2	Unknown
PartitionRecord	446	16*4	Array of four MBR partition records.
Signature	510	2	Must be 0xaa55.

The MBR contains four partition records that define the beginning and ending LBA addresses that a partition consumes on a hard disk. The partition record contains a legacy Cylinder Head Sector (CHS) address that is not used in EFI. EFI utilizes the starting LBA entry to define the starting LBA of the partition on the disk. The size of the partition is defined by the size in LBA field.

The boot indicator field is not used by EFI firmware. The operating system indicator value of 0xEF defines a partition that contains an EFI file system. The other values of the system indicator are not defined by this specification. If an MBR partition has an operating system indicator value of 0xEF, then the firmware must add the EFI System Partition GUID to the handle for the MBR partition using **`InstallProtocolInterface()`**. This will allow drivers and applications, including OS loaders, to easily search for handles that represent EFI System Partitions.

Table 11-6. Legacy Master Boot Record Partition Record

Mnemonic	Byte Offset	Byte Length	Description
Boot Indicator	0	1	Not used by EFI firmware. Set to 0x80 to indicate that this is the bootable legacy partition.
Start Head	1	1	Start of partition in CHS address, not used by EFI firmware.
Start Sector	2	1	Start of partition in CHS address, not used by EFI firmware.
Start Track	3	1	Start of partition in CHS address, not used by EFI firmware.
OS Type	4	1	OS type. A value of 0xEF defines an EFI system partition. Other values are reserved for legacy operating systems, and allocated independently of the EFI specification.
End head	5	1	End of partition in CHS address, not used by EFI firmware.
End Sector	6	1	End of partition in CHS address, not used by EFI firmware.
End Track	7	1	End of partition in CHS address, not used by EFI firmware.
Starting LBA	8	4	Starting LBA address of the partition on the disk. Used by EFI firmware to define the start of the partition.
Size In LBA	12	4	Size of partition in LBA. Used by EFI firmware to determine the size of the partition.

EFI defines a valid legacy MBR as follows. The signature at the end of the MBR must be 0xaa55. Each MBR partition record must be checked to make sure that the partition that it defines physically resides on the disk. Each partition record must be checked to make sure it does not overlap with other partition records. A partition record that contains an *OSIndicator* value of zero or a *SizeInLBA* value of zero may be ignored. If any of these checks fail, the MBR is not considered valid.

11.2.2.4 Legacy Master Boot Record and GPT Partitions

The GPT partition structure does not support nesting of partitions. However it is legal to have a legacy Master Boot Record nested inside a GPT partition.

On all GUID Partition Table disks a Protective MBR (PMBR) in the first LBA of the disk precedes the GUID Partition Table Header to maintain compatibility with existing tools that do not understand GPT partition structures. The Protective MBR has the same format as a legacy MBR, contains one partition entry of OS type 0xEE and reserves the entire space used on the disk by the GPT partitions, including all headers. The Protective MBR that precedes a GUID Partition Table Header is shown in Table 11-7. If the GPT partition is larger than a partition that can be represented by a legacy MBR, values of all *F*s must be used to signify that all space that can be possibly reserved by the MBR is being reserved.

Table 11-7. PMBR Entry to Precede a GUID Partition Table Header

Mnemonic	Byte Offset	Byte Length	Description
Boot Indicator	0	1	Must be set to zero to indicate nonbootable partition.
Start Head	1	1	Set to match the Starting LBA of the EFI Partition structure. Must be set to 0xFFFFFF if it is not possible to represent the starting LBA.
Start Sector	2	1	
Start Track	3	1	
OS Type	4	1	Must be 0xEE.
End head	5	1	Set to match the Ending LBA of the EFI Partition structure. Must be set to 0xFFFFFF if it is not possible to represent the starting LBA.
End Sector	6	1	
End Track	7	1	
Starting LBA	8	4	Must be 1 by definition.
Size In LBA	12	4	Length of EFI Partition Head, 0xFFFFFFFF if this value overflows.

11.2.3 Media Formats

This section describes how booting from different types of removable media is handled. In general the rules are consistent regardless of a media's physical type and whether it is removable or not.

11.2.3.1 Removable Media

Removable media may contain a standard FAT12, FAT16, or FAT32 file system. Legacy 1.44 MB floppy devices typically support a FAT12 file system.

Booting from a removable media device can be accomplished the same way as any other boot. The boot file path provided to the boot manager can consist of an EFI application image to load, or can merely be the path to a removable media device. In the first case, the path clearly indicates the image that is to be loaded. In the later case, the boot manager implements the policy to load the default application image from the device.

For removable media to be bootable under EFI, it must be built in accordance with the rules laid out in Section 3.4.1.1.

11.2.3.2 Diskette

EFI bootable diskettes follow the standard formatting conventions used on Intel architecture personal computers. The diskette contains only a single partition that complies to the EFI file system type. For diskettes to be bootable under EFI, it must be built in accordance with the rules laid out in Section 3.4.1.1.

Since the EFI file system definition does not use the code in the first block of the diskette, it is possible to boot Intel architecture personal computers using a diskette that is also formatted as an EFI bootable removable media device. The inclusion of boot code for Intel architecture personal computers is optional and not required by EFI.

Diskettes include the legacy 3.5-inch diskette drives as well as the newer larger capacity removable media drives such as an Iomega† Zip†, Fujitsu MO, or MKE LS-120/SuperDisk†.

11.2.3.3 Hard Drive

Hard drives may contain multiple partitions as defined in Section 11.2.2 on partition discovery. Any partition on the hard drive may contain a file system that the EFI firmware recognizes. Images that are to be booted must be stored under the EFI subdirectory as defined in Sections 11.2.1 and 11.2.2.

EFI code does not assume a fixed block size.

Since EFI firmware does not execute the MBR code and does not depend on the bootable flag field in the partition entry the hard disk can still boot and function normally on an Intel architecture-based personal computer.

11.2.3.4 CD-ROM and DVD-ROM

A CD-ROM or DVD-ROM may contain multiple partitions as defined Sections 11.2.1 and 11.2.2 and in the “El Torito” specification.

EFI code does not assume a fixed block size.

Since the EFI file system definition does not use the same Initial/Default entry as a legacy CD-ROM, it is possible to boot Intel architecture personal computers using an EFI CD-ROM or DVD-ROM. The inclusion of boot code for Intel architecture personal computers is optional and not required by EFI.

11.2.3.5 Network

To boot from a network device, the Boot Manager uses the Load File Protocol to perform a `LoadFile()` on the network device. This uses the PXE Base Code Protocol to perform DHCP and Discovery. This may result in a list of possible boot servers along with the boot files available on each server. The Load File Protocol for a network boot may then optionally produce a menu of these selections for the user to choose from. If this menu is presented, it will always have a timeout, so the Load File Protocol can automatically boot the default boot selection. If there is only one possible boot file, then the Load File Protocol can automatically attempt to load the one boot file.

The Load File Protocol will download the boot file using the MTFTP service in the PXE Base Code Protocol. The downloaded image must be an EFI image that the platform supports.

11.3 File System Protocol

This section defines the File System protocol. This protocol allows code running in the EFI boot services environment to obtain file based access to a device. The Simple File System protocol is used to open a device volume and return an **EFI_FILE** that provides interfaces to access files on a device volume.

Simple File System Protocol

Summary

Provides a minimal interface for file-type access to a device.

GUID

```
#define SIMPLE_FILE_SYSTEM_PROTOCOL \
{ 0964e5b22-6459-11d2-8e39-00a0c969723b }
```

Revision Number

```
#define EFI_FILE_IO_INTERFACE_REVISION   0x00010000
```

Protocol Interface Structure

```
typedef struct _EFI_FILE_IO_INTERFACE {
     UINT64                          Revision;
     EFI_VOLUME_OPEN                 OpenVolume;
} EFI_FILE_IO_INTERFACE;
```

Parameters

Revision	The version of the **EFI_FILE_IO_INTERFACE**. The version specified by this specification is 0x00010000. All future revisions must be backwards compatible. If a future version is not backwards compatible, it is not the same GUID.
OpenVolume	Opens the volume for file I/O access. See the **OpenVolume()** function description.

Description

The Simple File System protocol provides a minimal interface for file-type access to a device. This protocol is only supported on some devices.

Devices that support the Simple File System protocol return an **`EFI_FILE_IO_INTERFACE`**. The only function of this interface is to open a handle to the root directory of the file system on the volume. Once opened, all accesses to the volume are performed through the volume's file handles, using the **`EFI_FILE`** protocol. The volume is closed by closing all the open file handles.

The firmware automatically creates handles for any block device that supports the following file system formats:

- FAT12
- FAT16
- FAT32

EFI_FILE_IO_INTERFACE.OpenVolume()

Summary

Opens the root directory on a volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_VOLUME_OPEN) (
    IN EFI_FILE_IO_INTERFACE      *This,
    OUT EFI_FILE                  **Root
    );
```

Parameters

This	A pointer to the volume to open the root directory of. See the type **EFI_FILE_IO_INTERFACE** description.
Root	A pointer to the location to return the opened file handle for the root directory. See the type **EFI_FILE** protocol description.

Description

The **OpenVolume()** function opens a volume, and returns a file handle to the volume's root directory. This handle is used to perform all other file I/O operations. The volume remains open until all the file handles to it are closed.

If the medium is changed while there are open file handles to the volume, all file handles to the volume will return **EFI_MEDIA_CHANGED**. To access the files on the new medium, the volume must be reopened with **OpenVolume()**. If the new medium is a different file system than the one supplied in the **EFI_HANDLE**'s **DevicePath** for the Simple File System protocol, **OpenVolume()** will return **EFI_UNSUPPORTED**.

Status Codes Returned

EFI_SUCCESS	The file volume was opened.
EFI_UNSUPPORTED	The volume does not support the requested file system type.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_ACCESS_DENIED	The service denied access to the file.
EFI_OUT_OF_RESOURCES	The file volume was not opened.
EFI_MEDIA_CHANGED	The device has a different medium in it or the medium is no longer supported. Any existing file handles for this volume are no longer valid. To access the files on the new medium, the volume must be reopened with **OpenVolume()**.

11.4 EFI_FILE Protocol

The protocol and functions described in this section support access to EFI-supported file systems.

EFI_FILE Protocol

Summary

Provides file based access to supported file systems.

Revision Number

```
#define EFI_FILE_REVISION          0x00010000
```

Protocol Interface Structure

```
typedef struct _EFI_FILE {
     UINT64                      Revision;
     EFI_FILE_OPEN               Open;
     EFI_FILE_CLOSE              Close;
     EFI_FILE_DELETE             Delete;
     EFI_FILE_READ               Read;
     EFI_FILE_WRITE              Write;
     EFI_FILE_GET_POSITION       GetPosition;
     EFI_FILE_SET_POSITION       SetPosition;
     EFI_FILE_GET_INFO           GetInfo;
     EFI_FILE_SET_INFO           SetInfo;
     EFI_FILE_FLUSH              Flush;
} EFI_FILE;
```

Parameters

Revision	The version of the **EFI_FILE** interface. The version specified by this specification is 0x00010000. Future versions are required to be backward compatible to version 1.0.
Open	Opens or creates a new file. See the **Open()** function description.
Close	Closes the current file handle. See the **Close()** function description.
Delete	Deletes a file. See the **Delete()** function description.
Read	Reads bytes from a file. See the **Read()** function description.
Write	Writes bytes to a file. See the **Write()** function description.
GetPosition	Returns the current file position. See the **GetPosition()** function description.

SetPosition	Sets the current file position. See the **SetPosition()** function description.
GetInfo	Gets the requested file or volume information. See the **GetInfo()** function description.
SetInfo	Sets the requested file information. See the **SetInfo()** function description.
Flush	Flushes all modified data associated with the file to the device. See the **Flush()** function description.

Description

The **EFI_FILE** provides file IO access to supported file systems.

An **EFI_FILE** provides access to a file's or directory's contents, and is also a reference to a location in the directory tree of the file system in which the file resides. With any given file handle, other files may be opened relative to this file's location, yielding new file handles.

On requesting the file system protocol on a device, the caller gets the **EFI_FILE_IO_INTERFACE** to the volume. This interface is used to open the root directory of the file system when needed. The caller must **Close()** the file handle to the root directory, and any other opened file handles before exiting. While there are open files on the device, usage of underlying device protocol(s) that the file system is abstracting must be avoided. For example, when a file system that is layered on a **DISK_IO** / **BLOCK_IO** protocol, direct block access to the device for the blocks that comprise the file system must be avoided while there are open file handles to the same device.

A file system driver may cache data relating to an open file. A **Flush()** function is provided that flushes all dirty data in the file system, relative to the requested file, to the physical medium. If the underlying device may cache data, the file system must inform the device to flush as well.

EFI_FILE.Open()

Summary

Opens a new file relative to the source file's location.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_FILE_OPEN) (
     IN EFI_FILE              *This,
     OUT EFI_FILE             **NewHandle,
     IN CHAR16                *FileName,
     IN UINT64                OpenMode,
     IN UINT64                Attributes
     );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the file handle to the source location. This would typically be an open handle to a directory. See the type **EFI_FILE** protocol description.
NewHandle	A pointer to the location to return the opened handle for the new file. See the type **EFI_FILE** protocol description.
FileName	The Null-terminated string of the name of the file to be opened. The file name may contain the following path modifiers: "\", ".", and "..".
OpenMode	The mode to open the file. The only valid combinations that the file may be opened with are: Read, Read/Write, or Create/Read/Write. See "Related Definitions" below.
Attributes	Only valid for **EFI_FILE_MODE_CREATE**, in which case these are the attribute bits for the newly created file. See "Related Definitions" below.

Related Definitions

```
//*******************************************************
// Open Modes
//*******************************************************
#define EFI_FILE_MODE_READ          0x0000000000000001
#define EFI_FILE_MODE_WRITE         0x0000000000000002
#define EFI_FILE_MODE_CREATE        0x8000000000000000

//*******************************************************
// File Attributes
//*******************************************************
#define EFI_FILE_READ_ONLY          0x0000000000000001
#define EFI_FILE_HIDDEN             0x0000000000000002
#define EFI_FILE_SYSTEM             0x0000000000000004
```

```
#define EFI_FILE_RESERVED           0x0000000000000008
#define EFI_FILE_DIRECTORY          0x0000000000000010
#define EFI_FILE_ARCHIVE            0x0000000000000020
#define EFI_FILE_VALID_ATTR         0x0000000000000037
```

Description

The **Open()** function opens the file or directory referred to by *FileName* relative to the location of *This* and returns a *NewHandle*. The *FileName* may include the following path modifiers:

"\"	If the filename starts with a "\" the relative location is the root directory that *This* residues on; otherwise "\" separates name components. Each name component is opened in turn, and the handle to the last file opened is returned.
"."	Opens the current location.
".."	Opens the parent directory for the current location. If the location is the root directory the request will return an error, as there is no parent directory for the root directory.

If **EFI_FILE_MODE_CREATE** is set, then the file is created in the directory. If the final location of *FileName* does not refer to a directory, then the operation fails. If the file does not exist in the directory, then a new file is created. If the file already exists in the directory, then the existing file is opened.

If the medium of the device changes, all accesses (including the File handle) will result in **EFI_MEDIA_CHANGED**. To access the new medium, the volume must be reopened.

Status Codes Returned

EFI_SUCCESS	The file was opened.
EFI_NOT_FOUND	The specified file could not be found on the device.
EFI_NO_MEDIA	The device has no medium.
EFI_MEDIA_CHANGED	The device has a different medium in it or the medium is no longer supported.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	An attempt was made to create a file, or open a file for write when the media is write protected.
EFI_ACCESS_DENIED	The service denied access to the file.
EFI_OUT_OF_RESOURCES	Not enough resources were available to open the file.
EFI_VOLUME_FULL	The volume is full.

EFI_FILE.Close()

Summary

Closes a specified file handle.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_FILE_CLOSE) (
     IN EFI_FILE              *This
     );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the file handle to close. See the type **EFI_FILE** protocol description.

Description

The **Close()** function closes a specified file handle. All "dirty" cached file data is flushed to the device, and the file is closed. *In all cases the handle is closed.*

Status Codes Returned

EFI_SUCCESS	The file was closed.

EFI_FILE.Delete()

Summary

Closes and deletes a file.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_FILE_DELETE) (
      IN EFI_FILE            *This
      );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the handle to the file to delete. See the type **EFI_FILE** protocol description.

Description

The **Delete()** function closes and deletes a file. *In all cases the file handle is closed.* If the file cannot be deleted, the warning code **EFI_WARN_DELETE_FAILURE** is returned, but the handle is still closed.

Status Codes Returned

EFI_SUCCESS	The file was closed and deleted, and the handle was closed.
EFI_WARN_DELETE_FAILURE	The handle was closed, but the file was not deleted.

EFI_FILE.Read()

Summary

Reads data from a file.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_FILE_READ) (
      IN EFI_FILE            *This,
      IN OUT UINTN           *BufferSize,
      OUT VOID               *Buffer
      );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the file handle to read data from. See the type **EFI_FILE** protocol description.
BufferSize	On input, the size of the *Buffer*. On output, the amount of data returned in *Buffer*. In both cases, the size is measured in bytes.
Buffer	The buffer into which the data is read.

Description

The **Read()** function reads data from a file.

If *This* is not a directory, the function reads the requested number of bytes from the file at the file's current position and returns them in *Buffer*. If the read goes beyond the end of the file, the read length is truncated to the end of the file. The file's current position is increased by the number of bytes returned.

If *This* is a directory, the function reads the directory entry at the file's current position and returns the entry in *Buffer*. If the *Buffer* is not large enough to hold the current directory entry, then **EFI_BUFFER_TOO_SMALL** is returned and the current file position is *not* updated. *BufferSize* is set to be the size of the buffer needed to read the entry. On success, the current position is updated to the next directory entry. If there are no more directory entries, the read returns a zero-length buffer. **EFI_FILE_INFO** is the structure returned as the directory entry.

Status Codes Returned

EFI_SUCCESS	The data was read.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_BUFFER_TOO_SMALL	The *BufferSize* is too small to read the current directory entry. *BufferSize* has been updated with the size needed to complete the request.

EFI_FILE.Write()

Summary

Writes data to a file.

```
EFI_STATUS
(EFIAPI *EFI_FILE_WRITE) (
      IN EFI_FILE            *This,
      IN OUT UINTN           *BufferSize,
      IN VOID                *Buffer
      );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the file handle to write data to. See the type **EFI_FILE** protocol description.
BufferSize	On input, the size of the *Buffer*. On output, the amount of data actually written. In both cases, the size is measured in bytes.
Buffer	The buffer of data to write.

Description

The **Write()** function writes the specified number of bytes to the file at the current file position. The current file position is advanced the actual number of bytes written, which is returned in *BufferSize*. Partial writes only occur when there has been a data error during the write attempt (such as "file space full"). The file is automatically grown to hold the data if required.

Direct writes to opened directories are not supported.

Status Codes Returned

EFI_SUCCESS	The data was written.
EFI_UNSUPPORT	Writes to open directory files are not supported.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	The file or medium is write protected.
EFI_ACCESS_DENIED	The file was opened read only.
EFI_VOLUME_FULL	The volume is full.

EFI_FILE.SetPosition()

Summary

Sets a file's current position.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_FILE_SET_POSITION) (
      IN EFI_FILE              *This,
      IN UINT64                Position
      );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the he file handle to set the requested position on. See the type **EFI_FILE** protocol description.
Position	The byte position from the start of the file to set.

Description

The **SetPosition()** function sets the current file position for the handle to the position supplied. With the exception of seeking to position 0xFFFFFFFFFFFFFFFF, only absolute positioning is supported, and seeking past the end of the file is allowed (a subsequent write would grow the file). Seeking to position 0xFFFFFFFFFFFFFFFF causes the current position to be set to the end of the file.

If *This* is a directory, the only position that may be set is zero. This has the effect of starting the read process of the directory entries over.

Status Codes Returned

EFI_SUCCESS	The position was set.
EFI_UNSUPPORTED	The seek request for nonzero is not valid on open directories.

EFI_FILE.GetPosition()

Summary

Returns a file's current position.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_GET_POSITION) (
      IN EFI_FILE              *This,
      OUT UINT64               *Position
      );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the file handle to get the current position on. See the type **EFI_FILE** protocol description.
Position	The address to return the file's current position value.

Description

The **GetPosition()** function returns the current file position for the file handle. For directories, the current file position has no meaning outside of the file system driver and as such the operation is not supported. An error is returned if *This* is a directory.

Status Codes Returned

EFI_SUCCESS	The position was returned.
EFI_UNSUPPORTED	The request is not valid on open directories.

EFI_FILE.GetInfo()

Summary

Returns information about a file.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_FILE_GET_INFO) (
      IN EFI_FILE             *This,
      IN EFI_GUID             *InformationType,
      IN OUT UINTN            *BufferSize,
      OUT VOID                *Buffer
      );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the file handle the requested information is for. See the type **EFI_FILE** protocol description.
InformationType	The type identifier for the information being requested. Type **EFI_GUID** is defined in Chapter 5. See the **EFI_FILE_INFO** and **EFI_FILE_SYSTEM_INFO** descriptions for the related GUID definitions.
BufferSize	On input, the size of *Buffer*. On output, the amount of data returned in *Buffer*. In both cases, the size is measured in bytes.
Buffer	A pointer to the data buffer to return. The buffer's type is indicated by *InformationType*.

Description

The **GetInfo()** function returns information of type *InformationType* for the requested file. If the file does not support the requested information type, then **EFI_UNSUPPORTED** is returned. If the buffer is not large enough to fit the requested structure, **EFI_BUFFER_TOO_SMALL** is returned and the *BufferSize* is set to the size of buffer that is required to make the request.

The information types defined by this specification are required information types that all file systems must support.

Status Codes Returned

EFI_SUCCESS	The information was set.
EFI_UNSUPPORTED	The *InformationType* is not known.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_BUFFER_TOO_SMALL	The *BufferSize* is too small to read the current directory entry. *BufferSize* has been updated with the size needed to complete the request.

EFI_FILE.SetInfo()

Summary

Sets information about a file.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_FILE_SET_INFO) (
     IN EFI_FILE                  *This,
     IN EFI_GUID                  *InformationType,
     IN UINTN                     BufferSize,
     IN VOID                      *Buffer
     );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the file handle the information is for. See the type **EFI_FILE** protocol description.
InformationType	The type identifier for the information being set. Type **EFI_GUID** is defined in Chapter 5. See the **EFI_FILE_INFO** and **EFI_FILE_SYSTEM_INFO** descriptions for the related GUID definitions.
BufferSize	The size, in bytes, of *Buffer*.
Buffer	A pointer to the data buffer to write. The buffer's type is indicated by *InformationType*.

Description

The **SetInfo()** function sets information of type *InformationType* on the requested file.

Status Codes Returned

EFI_SUCCESS	The information was set.
EFI_UNSUPPORTED	The *InformationType* is not known.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	The file or medium is write protected.
EFI_ACCESS_DENIED	The file was opened read-only.
EFI_VOLUME_FULL	The volume is full.
EFI_BAD_BUFFER_SIZE	*BufferSize* is smaller than the size of the type indicated by *InformationType*.

EFI_FILE.Flush()

Summary

Flushes all modified data associated with a file to a device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_FILE_FLUSH) (
     IN EFI_FILE           *This
     );
```

Parameters

This	A pointer to the **EFI_FILE** instance that is the file handle to flush. See the type **EFI_FILE** protocol description.

Description

The **Flush()** function flushes all modified data associated with a file to a device.

Status Codes Returned

EFI_SUCCESS	The data was flushed.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	The file or medium is write protected.
EFI_ACCESS_DENIED	The file was opened read-only.
EFI_VOLUME_FULL	The volume is full.

EFI_FILE_INFO

Summary

Provides a GUID and a data structure that can be used with **EFI_FILE.SetInfo()** and **EFI_FILE.GetInfo()** to set or get generic file information.

GUID

```
#define EFI_FILE_INFO_ID \
{ 09576e92-6d3f-11d2-8e39-00a0c969723b }
```

Related Definitions

```
typedef struct {
     UINT64                  Size;
     UINT64                  FileSize;
     UINT64                  PhysicalSize;
     EFI_TIME                CreateTime;
     EFI_TIME                LastAccessTime;
     EFI_TIME                ModificationTime;
     UINT64                  Attribute;
     CHAR16                  FileName[];
} EFI_FILE_INFO;

//*******************************************************
// File Attribute Bits
//*******************************************************

#define EFI_FILE_READ_ONLY       0x0000000000000001
#define EFI_FILE_HIDDEN          0x0000000000000002
#define EFI_FILE_SYSTEM          0x0000000000000004
#define EFI_FILE_RESERVED        0x0000000000000008
#define EFI_FILE_DIRECTORY       0x0000000000000010
#define EFI_FILE_ARCHIVE         0x0000000000000020
#define EFI_FILE_VALID_ATTR      0x0000000000000037
```

Parameters

Size	Size of the **EFI_FILE_INFO** structure, including the Null-terminated Unicode *FileName* string.
FileSize	The size of the file in bytes.
PhysicalSize	The amount of physical space the file consumes on the file system volume.
CreateTime	The time the file was created.
LastAccessTime	The time when the file was last accessed.
ModificationTime	The time when the file's contents were last modified.
Attribute	The attribute bits for the file. See "Related Definitions" above.
FileName	The Null-terminated Unicode name of the file.

Description

The **EFI_FILE_INFO** data structure supports **GetInfo()** and **SetInfo()** requests. In the case of **SetInfo()**, the following additional rules apply:

- On directories, the file size is determined by the contents of the directory and cannot be changed by setting *FileSize*. On directories, *FileSize* is ignored during a **SetInfo()**.
- The *PhysicalSize* is determined by the *FileSize* and cannot be changed. This value is ignored during a **SetInfo()** request.
- The **EFI_FILE_DIRECTORY** attribute bit cannot be changed. It must match the file's actual type.
- A value of zero in *CreateTime*, *LastAccess*, or *ModificationTime* causes the fields to be ignored (and not updated).

EFI_FILE_SYSTEM_INFO

Summary

Provides a GUID and a data structure that can be used with **EFI_FILE.GetInfo()** to get information about the system volume, and **EFI_FILE.SetInfo()** to set the system volume's volume label.

GUID

```
#define EFI_FILE_SYSTEM_INFO_ID \
{ 09576e93-6d3f-11d2-8e39-00a0c969723b }
```

Related Definitions

```
typedef struct {
      UINT64                        Size;
      BOOLEAN                       ReadOnly;
      UINT64                        VolumeSize;
      UINT64                        FreeSpace;
      UINT32                        BlockSize;
      CHAR16                        VolumeLabel[];
} EFI_FILE_SYSTEM_INFO;
```

Parameters

Size	Size of the **EFI_FILE_SYSTEM_INFO** structure, including the Null-terminated Unicode *VolumeLabel* string.
ReadOnly	**TRUE** if the volume only supports read access.
VolumeSize	The number of bytes managed by the file system.
FreeSpace	The number of available bytes for use by the file system.
BlockSize	The nominal block size by which files are typically grown.
VolumeLabel	The Null-terminated string that is the volume's label.

Description

The **EFI_FILE_SYSTEM_INFO** data structure is an information structure that can be obtained on the root directory file handle. The root directory file handle is the file handle first obtained on the initial call to the **HandleProtocol()** function to open the file system interface. All of the fields are read-only except for *VolumeLabel*. The system volume's *VolumeLabel* can be created or modified by calling **EFI_FILE.SetInfo()** with an updated *VolumeLabel* field.

EFI_FILE_SYSTEM_VOLUME_LABEL

Summary

Provides a GUID and a data structure that can be used with **EFI_FILE.GetInfo()** or **EFI_FILE.SetInfo()** to get or set information about the system's volume label.

GUID

```
#define EFI_FILE_SYSTEM_VOLUME_LABEL_ID \
      { DB47D7D3-FE81-11d3-9A35-0090273FC14D }
```

Related Definitions

```
typedef struct {
      CHAR16                      VolumeLabel[];
} EFI_FILE_SYSTEM_VOLUME_LABEL;
```

Parameters

VolumeLabel	The Null-terminated string that is the volume's label.

Description

The **EFI_FILE_SYSTEM_VOLUME_LABEL** data structure is an information structure that can be obtained on the root directory file handle. The root directory file handle is the file handle first obtained on the initial call to the **HandleProtocol()** function to open the file system interface. The system volume's *VolumeLabel* can be created or modified by calling **EFI_FILE.SetInfo()** with an updated *VolumeLabel* field.

11.5 DISK_IO Protocol

This section defines the Disk I/O protocol. This protocol is used to abstract the block accesses of the Block I/O protocol to a more general offset-length protocol. The firmware is responsible for adding this protocol to any Block I/O interface that appears in the system that does not already have a Disk I/O protocol. File systems and other disk access code utilize the Disk I/O protocol.

DISK_IO Protocol

Summary

This protocol is used to abstract Block I/O interfaces.

GUID

```
#define DISK_IO_PROTOCOL    \
     { CE345171-BA0B-11d2-8e4F-00a0c969723b }
```

Revision Number

```
#define EFI_DISK_IO_INTERFACE_REVISION      0x00010000
```

Protocol Interface Structure

```
typedef struct _EFI_DISK_IO {
     UINT64                         Revision;
     EFI_DISK_READ                  ReadDisk;
     EFI_DISK_WRITE                 WriteDisk;
} EFI_DISK_IO;
```

Parameters

Revision	The revision to which the disk I/O interface adheres. All future revisions must be backwards compatible. If a future version is not backwards compatible, it is not the same GUID.
ReadDisk	Reads data from the disk. See the **ReadDisk()** function description.
WriteDisk	Writes data to the disk. See the **WriteDisk()** function description.

Description

The **EFI_DISK_IO** protocol is used to control block I/O interfaces.

The disk I/O functions allow I/O operations that need not be on the underlying device's block boundaries or alignment requirements. This is done by copying the data to/from internal buffers as needed to provide the proper requests to the block I/O device. Outstanding write buffer data is flushed by using the **Flush()** function of the **EFI_BLOCK_IO** protocol on the device handle.

The firmware automatically adds an **EFI_DISK_IO** interface to any **EFI_BLOCK_IO** interface that is produced. It also adds file system, or logical block I/O, interfaces to any **EFI_DISK_IO** interface that contains any recognized file system or logical block I/O devices. The firmware must automatically support the following required formats:

- The EFI FAT12, FAT16, and FAT32 file system type.
- The legacy master boot record partition block. (The presence of this on any block I/O device is optional, but if it is present the firmware is responsible for allocating a logical device for each partition).
- The extended partition record partition block.
- The El Torito logical block devices.

EFI_DISK_IO.ReadDisk()

Summary

Reads a specified number of bytes from a device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_DISK_READ) (
      IN EFI_DISK_IO                    *This,
      IN UINT32                         MediaId,
      IN UINT64                         Offset,
      IN UINTN                          BufferSize,
      OUT VOID                          *Buffer
      );
```

Parameters

This	Indicates a pointer to the calling context. Type **EFI_DISK_IO** is defined in the **DISK_IO** protocol description.
MediaId	ID of the medium to be read.
Offset	The starting byte offset on the logical block I/O device to read from.
BufferSize	The size in bytes of *Buffer*. The number of bytes to read from the device.
Buffer	A pointer to the destination buffer for the data. The caller is responsible for either having implicit or explicit ownership of the buffer.

Description

The **ReadDisk()** function reads the number of bytes specified by *BufferSize* from the device. All the bytes are read, or an error is returned. If there is no medium in the device, the function returns **EFI_NO_MEDIA**. If the *MediaId* is not the ID of the medium currently in the device, the function returns **EFI_MEDIA_CHANGED**.

Status Codes Returned

EFI_SUCCESS	The data was read correctly from the device.
EFI_DEVICE_ERROR	The device reported an error while performing the read operation.
EFI_NO_MEDIA	There is no medium in the device.
EFI_MEDIA_CHANGED	The *MediaId* is not for the current medium.
EFI_INVALID_PARAMETER	The read request contains device addresses that are not valid for the device.

EFI_DISK_IO.WriteDisk()

Summary

Writes a specified number of bytes to a device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_DISK_WRITE) (
      IN EFI_DISK_IO                *This,
      IN UINT32                     MediaId,
      IN UINT64                     Offset,
      IN UNITN                      BufferSize,
      IN VOID                       *Buffer
      );
```

Parameters

This	Indicates a pointer to the calling context. Type **EFI_DISK_IO** is defined in the **DISK_IO** protocol description.
MediaId	ID of the medium to be written.
Offset	The starting byte offset on the logical block I/O device to write.
BufferSize	The size in bytes of *Buffer*. The number of bytes to write to the device.
Buffer	A pointer to the buffer containing the data to be written.

Description

The **WriteDisk()** function writes the number of bytes specified by *BufferSize* to the device. All bytes are written, or an error is returned. If there is no medium in the device, the function returns **EFI_NO_MEDIA**. If the *MediaId* is not the ID of the medium currently in the device, the function returns **EFI_MEDIA_CHANGED**.

Status Codes Returned

EFI_SUCCESS	The data was written correctly to the device.
EFI_WRITE_PROTECTED	The device cannot be written to.
EFI_NO_MEDIA	There is no medium in the device.
EFI_MEDIA_CHANGED	The *MediaId* is not for the current medium.
EFI_DEVICE_ERROR	The device reported an error while performing the write operation.
EFI_INVALID_PARAMETER	The write request contains device addresses that are not valid for the device.

11.6 BLOCK_IO Protocol

This chapter defines the Block I/O protocol. This protocol is used to abstract mass storage devices to allow code running in the EFI boot services environment to access them without specific knowledge of the type of device or controller that manages the device. Functions are defined to read and write data at a block level from mass storage devices as well as to manage such devices in the EFI boot services environment.

BLOCK_IO Protocol

Summary

This protocol provides control over block devices.

GUID

```
#define BLOCK_IO_PROTOCOL   \
     { 964e5b21-6459-11d2-8e39-00a0c969723b }
```

Revision Number

```
#define EFI_BLOCK_IO_INTERFACE_REVISION    0x00010000
```

Protocol Interface Structure

```
typedef struct _EFI_BLOCK_IO {
    UINT64                  Revision;

    EFI_BLOCK_IO_MEDIA      *Media;

    EFI_BLOCK_RESET         Reset;
    EFI_BLOCK_READ          ReadBlocks;
    EFI_BLOCK_WRITE         WriteBlocks;
    EFI_BLOCK_FLUSH         FlushBlocks;
} EFI_BLOCK_IO;
```

Parameters

Revision	The revision to which the block IO interface adheres. All future revisions must be backwards compatible. If a future version is not back wards compatible it is not the same GUID.
Media	A pointer to the **EFI_BLOCK_IO_MEDIA** data for this device. Type **EFI_BLOCK_IO_MEDIA** is defined in “Related Definitions” below.
Reset	Resets the block device hardware. See the **Reset()** function description.
ReadBlocks	Reads the requested number of blocks from the device. See the **ReadBlocks()** function description.

WriteBlocks	Writes the requested number of blocks to the device. See the **WriteBlocks()** function description.
FlushBlocks	Flushes and cache blocks. This function is optional and only needs to be supported on block devices that cache writes. See the **FlushBlocks()** function description.

Related Definitions

```
//*******************************************************
// EFI_BLOCK_IO_MEDIA
//*******************************************************

typedef struct {
    UINT32              MediaId;
    BOOLEAN             RemovableMedia;
    BOOLEAN             MediaPresent;

    BOOLEAN             LogicalPartition;
    BOOLEAN             ReadOnly;
    BOOLEAN             WriteCaching;

    UINT32              BlockSize;
    UINT32              IoAlign;

    EFI_LBA             LastBlock;
} EFI_BLOCK_IO_MEDIA;

//*******************************************************
// EFI_LBA
//*******************************************************

typedef UINT64          EFI_LBA;
```

The following data values in **EFI_BLOCK_IO_MEDIA** are read-only and are updated by the code that produces the **EFI_BLOCK_IO** protocol functions:

MediaId	The current media ID. If the media changes, this value is changed.
RemovableMedia	**TRUE** if the media is removable; otherwise, **FALSE**.
MediaPresent	**TRUE** if there is a media currently present in the device; otherwise, **FALSE**. This field shows the media present status as of the most recent **ReadBlocks()** or **WriteBlocks()** call.
LogicalPartition	**TRUE** if LBA 0 is the first block of a partition; otherwise **FALSE**. For media with only one partition this would be **TRUE**.

ReadOnly	**TRUE** if the media is marked read-only otherwise, **FALSE**. This field shows the read-only status as of the most recent **WriteBlocks()** call.
WriteCaching	**TRUE** if the **WriteBlocks()** function caches write data.
BlockSize	The intrinsic block size of the device. If the media changes, then this field is updated.
IoAlign	Supplies the alignment requirement for any buffer used in a data transfer. *IoAlign* values of 0 and 1 mean that the buffer can be placed anywhere in memory. Otherwise, *IoAlign* must be a power of 2, and the requirement is that the start address of a buffer must be evenly divisible by *IoAlign* with no remainder.
LastBlock	The last logical block address on the device. If the media changes, then this field is updated.

Description

The *LogicalPartition* is **TRUE** if the device handle is for a partition. For media that have only one partition, the value will always be **TRUE**. For media that have multiple partitions, this value is **FALSE** for the handle that accesses the entire device. The firmware is responsible for adding device handles for each partition on such media.

The firmware is responsible for adding an **EFI_DISK_IO** interface to every **EFI_BLOCK_IO** interface in the system. The **EFI_DISK_IO** interface allows byte-level access to devices.

EFI_BLOCK_IO.Reset()

Summary

Resets the block device hardware.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_BLOCK_RESET) (
     IN EFI_BLOCK_IO              *This,
     IN BOOLEAN                   ExtendedVerification
     );
```

Parameters

This	Indicates a pointer to the calling context. Type **EFI_BLOCK_IO** is defined in the **BLOCK_IO** protocol description.
ExtendedVerification	Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description

The **Reset()** function resets the block device hardware.

As part of the initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the *ExtendedVerification* flag is **TRUE** the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware and/or EFI driver to implement.

Status Codes Returned

EFI_SUCCESS	The block device was reset.
EFI_DEVICE_ERROR	The block device is not functioning correctly and could not be reset.

EFI_BLOCK_IO.ReadBlocks()

Summary

Reads the requested number of blocks from the device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_BLOCK_READ) (
     IN EFI_BLOCK_IO                 *This,
     IN UINT32                       MediaId,
     IN EFI_LBA                      LBA,
     IN UINTN                        BufferSize,
     OUT VOID                        *Buffer
     );
```

Parameters

This	Indicates a pointer to the calling context. Type **EFI_BLOCK_IO** is defined in the **BLOCK_IO** protocol description.
MediaId	The media ID that the read request is for.
LBA	The starting logical block address to read from on the device. Type **EFI_LBA** is defined in the **BLOCK_IO** protocol description.
BufferSize	The size of the *Buffer* in bytes. This must be a multiple of the intrinsic block size of the device.
Buffer	A pointer to the destination buffer for the data. The caller is responsible for either having implicit or explicit ownership of the buffer.

Description

The **ReadBlocks()** function reads the requested number of blocks from the device. All the blocks are read, or an error is returned.

If there is no media in the device, the function returns **EFI_NO_MEDIA**. If the *MediaId* is not the ID for the current media in the device, the function returns **EFI_MEDIA_CHANGED**.

Status Codes Returned

EFI_SUCCESS	The data was read correctly from the device.
EFI_DEVICE_ERROR	The device reported an error while attempting to perform the read operation.
EFI_NO_MEDIA	There is no media in the device.
EFI_MEDIA_CHANGED	The *MediaId* is not for the current media.
EFI_BAD_BUFFER_SIZE	The *BufferSize* parameter is not a multiple of the intrinsic block size of the device.
EFI_INVALID_PARAMETER	The read request contains LBAs that are not valid, or the buffer is not on proper alignment.

EFI_BLOCK_IO.WriteBlocks()

Summary

Writes a specified number of blocks to the device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_BLOCK_WRITE) (
      IN EFI_BLOCK_IO                 *This,
      IN UINT32                       MediaId,
      IN EFI_LBA                      LBA,
      IN UINTN                        BufferSize,
      IN VOID                         *Buffer
      );
```

Parameters

This	Indicates a pointer to the calling context. Type **EFI_BLOCK_IO** is defined in the **BLOCK_IO** protocol description.
MediaId	The media ID that the write request is for.
LBA	The starting logical block address to be written. The caller is responsible for writing to only legitimate locations. Type **EFI_LBA** is defined in the **BLOCK_IO** protocol description.
BufferSize	The size in bytes of *Buffer*. This must be a multiple of the intrinsic block size of the device.
Buffer	A pointer to the source buffer for the data.

Description

The **WriteBlocks()** function writes the requested number of blocks to the device. All blocks are written, or an error is returned.

If there is no media in the device, the function returns **EFI_NO_MEDIA**. If the *MediaId* is not the ID for the current media in the device, the function returns **EFI_MEDIA_CHANGED**.

Status Codes Returned

EFI_SUCCESS	The data were written correctly to the device.
EFI_WRITE_PROTECTED	The device cannot be written to.
EFI_NO_MEDIA	There is no media in the device.
EFI_MEDIA_CHANGED	The *MediaId* is not for the current media.
EFI_DEVICE_ERROR	The device reported an error while attempting to perform the write operation.
EFI_BAD_BUFFER_SIZE	The *BufferSize* parameter is not a multiple of the intrinsic block size of the device.
EFI_INVALID_PARAMETER	The write request contains LBAs that are not valid, or the buffer is not on proper alignment.

EFI_BLOCK_IO.FlushBlocks()

Summary

Flushes all modified data to a physical block device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_BLOCK_FLUSH) (
      IN EFI_BLOCK_IO              *This
      );
```

Parameters

This	Indicates a pointer to the calling context. Type **EFI_BLOCK_IO** is defined in the **BLOCK_IO** protocol description.

Description

The **FlushBlocks()** function flushes all modified data to the physical block device.

All data written to the device prior to the flush must be physically written before returning **EFI_SUCCESS** from this function. This would include any cached data the driver may have cached, and cached data the device may have cached. Even if there were no outstanding data, a read request to a device with removable media following a flush will always cause a device access.

Status Codes Returned

EFI_SUCCESS	All outstanding data were written correctly to the device.
EFI_DEVICE_ERROR	The device reported an error while attempting to write data.
EFI_NO_MEDIA	There is no media in the device.

11.7 UNICODE_COLLATION Protocol

This section defines the Unicode Collation protocol. This protocol is used to allow code running in the boot services environment to perform lexical comparison functions on Unicode strings for given languages.

UNICODE_COLLATION Protocol

Summary

Is used to perform case-insensitive comparisons of Unicode strings.

GUID

```
#define UNICODE_COLLATION_PROTOCOL \
    { 1d85cd7f-f43d-11d2-9a0c-0090273fc14d }
```

Protocol Interface Structure

```
typedef struct {
      EFI_UNICODE_COLLATION_STRICOLL          StriColl;
      EFI_UNICODE_COLLATION_METAIMATCH        MetaiMatch;
      EFI_UNICODE_COLLATION_STRLWR            StrLwr;
      EFI_UNICODE_COLLATION_STRUPR            StrUpr;
      EFI_UNICODE_COLLATION_FATTOSTR          FatToStr;
      EFI_UNICODE_COLLATION_STRTOFAT          StrToFat;
      CHAR8                                   *SupportedLanguages;
} UNICODE_COLLATION_INTERFACE;
```

Parameters

StriColl	Performs a case-insensitive comparison of two Null-terminated Unicode strings. See the **StriColl()** function description.
MetaiMatch	Performs a case-insensitive comparison between a Null-terminated Unicode pattern string and a Null-terminated Unicode string. The pattern string can use the '?' wildcard to match any character, and the '*' wildcard to match any substring. See the **MetaiMatch()** function description.
StrLwr	Converts all the Unicode characters in a Null-terminated Unicode string to lowercase Unicode characters. See the **StrLwr()** function description.
StrUpr	Converts all the Unicode characters in a Null-terminated Unicode string to uppercase Unicode characters. See the **StrUpr()** function description.
FatToStr	Converts an 8.3 FAT file name using an OEM character set to a Null-terminated Unicode string. See the **FatToStr()** function description.

StrToFat	Converts a Null-terminated Unicode string to legal characters in a FAT filename using an OEM character set. See the **StrToFat()** function description.
SupportedLanguages	A Null-terminated ASCII string that contains one or more ISO 639-2 language codes.

Description

The **UNICODE_COLLATION** protocol is used to perform case-insensitive comparisons of Unicode strings.

One or more of the **UNICODE_COLLATION** protocols may be present at one time. Each protocol instance can support one or more language codes. The language codes that are supported in the **UNICODE_COLLATION** interface is declared in *SupportedLanguages*.

The *SupportedLanguages* field is a list of one or more 3-character language codes in a Null-terminated ASCII string. These language codes come from the ISO 639-2 Specification. For example, if the protocol supports English, then the string “eng” would be returned. If it supported both English and Spanish, then “engspa” would be returned.

The main motivation for this protocol is to help support file names in a file system driver. When a file is opened, a file name needs to be compared to the file names on the disk. In some cases, this comparison needs to be performed in a case-insensitive manner. In addition, this protocol can be used to sort files from a directory or to perform a case-insensitive file search.

UNICODE_COLLATION.StriColl()

Summary

Performs a case-insensitive comparison of two Null-terminated Unicode strings.

Prototype

```
INTN
(EFIAPI *EFI_UNICODE_COLLATION_STRICOLL) (
     IN UNICODE_COLLATION_INTERFACE   *This,
     IN CHAR16                        *s1,
     IN CHAR16                        *s2
     );
```

Parameters

This	A pointer to the **UNICODE_COLLATION_INTERFACE** instance. Type **UNICODE_COLLATION_INTERFACE** is defined in Section 11.7.
s1	A pointer to a Null-terminated Unicode string.
s2	A pointer to a Null-terminated Unicode string.

Description

The **StriColl()** function performs a case-insensitive comparison of two Null-terminated Unicode strings.

This function performs a case-insensitive comparison between the Unicode string *s1* and the Unicode string *s2* using the rules for the language codes that this protocol instance supports. If *s1* is equivalent to *s2*, then 0 is returned. If *s1* is lexically less than *s2*, then a negative number will be returned. If *s1* is lexically greater than *s2*, then a positive number will be returned. This function allows Unicode strings to be compared and sorted.

Status Codes Returned

0	s1 is equivalent to s2.
> 0	s1 is lexically greater than s2.
< 0	s1 is lexically less than s2.

UNICODE_COLLATION.MetaiMatch()

Summary

Performs a case-insensitive comparison of a Null-terminated Unicode pattern string and a Null-terminated Unicode string.

Prototype

```
BOOLEAN
(EFIAPI *EFI_UNICODE_COLLATION_METAIMATCH) (
      IN UNICODE_COLLATION_INTERFACE    *This,
      IN CHAR16                         *String,
      IN CHAR16                         *Pattern
      );
```

Parameters

This	A pointer to the **UNICODE_COLLATION_INTERFACE** instance. Type **UNICODE_COLLATION_INTERFACE** is defined in Section 11.7.
String	A pointer to a Null-terminated Unicode string.
Pattern	A pointer to a Null-terminated Unicode pattern string.

Description

The **MetaiMatch()** function performs a case-insensitive comparison of a Null-terminated Unicode pattern string and a Null-terminated Unicode string.

This function checks to see if the pattern of characters described by *Pattern* are found in *String*. The pattern check is a case-insensitive comparison using the rules for the language codes that this protocol instance supports. If the pattern match succeeds, then **TRUE** is returned. Otherwise **FALSE** is returned. The following syntax can be used to build the string *Pattern:*

*****	Match 0 or more characters.
?	Match any one character.
[*<char1><char2>…<charN>***]**	Match any character in the set.
[*<char1>***-***<char2>***]**	Match any character between <char1> and <char2>.
<char>	Match the character <char>.

Following is an example pattern for English:

`*.FW`	Matches all strings that end in ".FW" or ".fw" or ".Fw" or ".fW."
`[a-z]`	Match any letter in the alphabet.
`[!@#$%^&*()]`	Match any one of these symbols.
`z`	Match the character "z" or "Z."
`D?.*`	Match the character "D" or "d" followed by any character followed by a "." followed by any string.

Status Codes Returned

TRUE	Pattern was found in *String*.
FALSE	Pattern was not found in *String*.

UNICODE_COLLATION.StrLwr()

Summary

Converts all the Unicode characters in a Null-terminated Unicode string to lowercase Unicode characters.

Prototype

```
VOID
(EFIAPI *EFI_UNICODE_COLLATION_STRLWR) (
     IN UNICODE_COLLATION_INTERFACE   *This,
     IN OUT CHAR16                    *String
     );
```

Parameters

This	A pointer to the **UNICODE_COLLATION_INTERFACE** instance. Type **UNICODE_COLLATION_INTERFACE** is defined in Section 11.7.
String	A pointer to a Null-terminated Unicode string.

Description

This functions walks through all the Unicode characters in *String*, and converts each one to its lowercase equivalent if it has one. The converted string is returned in *String*.

UNICODE_COLLATION.StrUpr()

Summary

Converts all the Unicode characters in a Null-terminated Unicode string to uppercase Unicode characters.

Prototype

```
VOID
(EFIAPI *EFI_UNICODE_COLLATION_STRUPR) (
      IN UNICODE_COLLATION_INTERFACE   *This,
      IN OUT CHAR16                    *String
      );
```

Parameters

This	A pointer to the **UNICODE_COLLATION_INTERFACE** instance. Type **UNICODE_COLLATION_INTERFACE** is defined in Section 11.7.
String	A pointer to a Null-terminated Unicode string.

Description

This functions walks through all the Unicode characters in *String*, and converts each one to its uppercase equivalent if it has one. The converted string is returned in *String*.

UNICODE_COLLATION.FatToStr()

Summary

Converts an 8.3 FAT file name in an OEM character set to a Null-terminated Unicode string.

Prototype

```
VOID
(EFIAPI *EFI_UNICODE_COLLATION_FATTOSTR) (
     IN UNICODE_COLLATION_INTERFACE    *This,
     IN UINTN                          FatSize,
     IN CHAR8                          *Fat,
     OUT CHAR16                        *String
     );
```

Parameters

This	A pointer to the **UNICODE_COLLATION_INTERFACE** instance. Type **UNICODE_COLLATION_INTERFACE** is defined in Section 11.7.
FatSize	The size of the string *Fat* in bytes.
Fat	A pointer to a Null-terminated string that contains an 8.3 file name using an OEM character set.
String	A pointer to a Null-terminated Unicode string. The string must be preallocated to hold *FatSize* Unicode characters.

Description

This function converts the string specified by *Fat* with length *FatSize* to the Null-terminated Unicode string specified by *String*. The characters in *Fat* are from an OEM character set.

UNICODE_COLLATION.StrToFat()

Summary

Converts a Null-terminated Unicode string to legal characters in a FAT filename using an OEM character set.

Prototype

```
BOOLEAN
(EFIAPI *EFI_UNICODE_COLLATION_STRTOFAT) (
     IN UNICODE_COLLATION_INTERFACE   *This,
     IN CHAR16                        *String,
     IN UINTN                         FatSize,
     OUT CHAR8                        *Fat
     );
```

Parameters

This	A pointer to the **UNICODE_COLLATION_INTERFACE** instance. Type **UNICODE_COLLATION_INTERFACE** is defined in Section 11.7.
String	A pointer to a Null-terminated Unicode string.
FatSize	The size of the string *Fat* in bytes.
Fat	A pointer to a string that contains an 8.3 file name using an OEM character set.

Description

This function converts the first *FatSize* Unicode characters of *String* to the legal FAT characters in an OEM character set and stores then in the string *Fat*. The Unicode characters '.' (period) and ' ' (space) are ignored for this conversion. If no valid mapping from the Unicode character to a FAT character is available, then it is substituted with an '_'. This function returns **FALSE** if the return string *Fat* is an 8.3 file name. This function returns **TRUE** if the return string *Fat* is a Long File Name.

Status Codes Returned

TRUE	*Fat* is a Long File Name.
FALSE	*Fat* is an 8.3 file name.

12
Protocols - PCI Bus Support

12.1 PCI Root Bridge I/O Support

These sections (Sections 12.1 and 12.2) describe the PCI Root Bridge I/O Protocol. This protocol provides an I/O abstraction for a PCI Root Bridge that is produced by a PCI Host Bus Controller. A PCI Host Bus Controller is a hardware component that allows access to a group of PCI devices that share a common pool of PCI I/O and PCI Memory resources. This protocol is used by a PCI Bus Driver to perform PCI Memory, PCI I/O, and PCI Configuration cycles on a PCI Bus. It also provides services to perform different types of bus mastering DMA on a PCI bus. PCI device drivers will not directly use this protocol. Instead, they will use the I/O abstraction produced by the PCI Bus Driver. Only drivers that require direct access to the entire PCI bus should use this protocol. In particular, functions for managing PCI buses are defined here although other bus types may be supported in a similar fashion as extensions to this specification.

All the services described in this chapter that generate PCI transactions follow the ordering rules defined in the *PCI Specification*. If the processor is performing a combination of PCI transactions and system memory transactions, then there is no guarantee that the system memory transactions will be strongly ordered with respect to the PCI transactions. If strong ordering is required, then processor-specific mechanisms may be required to guarantee strong ordering. For example, Itanium-based systems may require the use of memory fences to guarantee ordering.

12.1.1 PCI Root Bridge I/O Overview

The interfaces provided in the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** are for performing basic operations to memory, I/O, and PCI configuration space. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources.

The **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** allows for future innovation of the platform. It abstracts device-specific code from the system memory map. This allows system designers to make changes to the system memory map without impacting platform independent code that is consuming basic system resources.

A platform can be viewed as a set of processors and a set of core chipset components that may produce one or more host buses. Figure 12-1 shows a platform with ***n*** processors (*CPU*s in the figure), and a set of core chipset components that produce ***m*** host bridges.

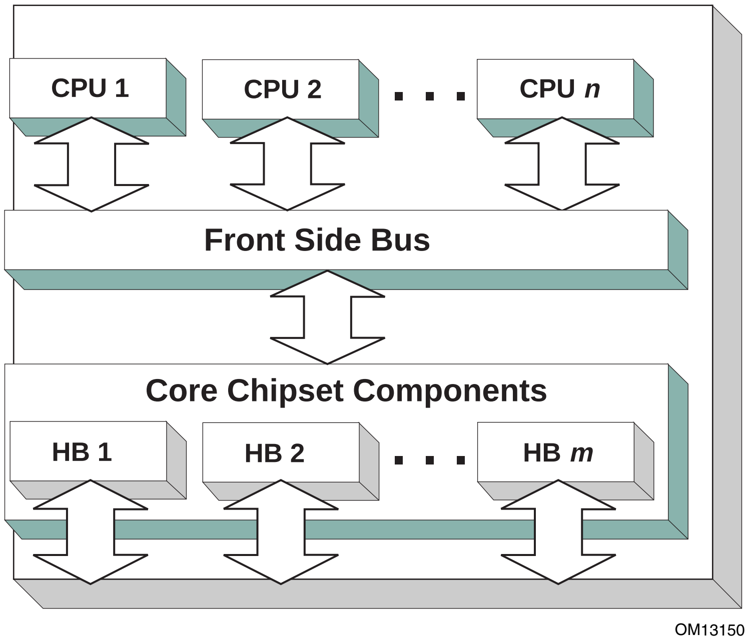


Figure 12-1. Host Bus Controllers

Simple systems with one PCI Host Bus Controller will contain a single instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. More complex system may contain multiple instances of this protocol. It is important to note that there is no relationship between the number of chipset components in a platform and the number of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** instances. This protocol abstracts access to a PCI Root Bridge from a software point of view, and it is attached to a device handle that represents a PCI Root Bridge. A PCI Root Bridge is a chipset component(s) that produces a physical PCI Bus. It is also the parent to a set of PCI devices that share common PCI I/O, PCI Memory, and PCI Prefetchable Memory regions. A PCI Host Bus Controller is composed of one or more PCI Root Bridges.

A PCI Host Bridge and PCI Root Bridge are different than a PCI Segment. A PCI Segment is a collection of up to 256 PCI busses that share the same PCI Configuration Space. Depending on the chipset, a single **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** may abstract a portion of a PCI Segment, or an entire PCI Segment. A PCI Host Bridge may produce one or more PCI Root Bridges. When a PCI Host Bridge produces multiple PCI Root Bridges, it is possible to have more than one PCI Segment.

PCI Root Bridge I/O Protocol instances are either produced by the system firmware or by an EFI driver. When a PCI Root Bridge I/O Protocol is produced, it is placed on a device handle along with an EFI Device Path Protocol instance. Figure 12-2 shows a sample device handle for a PCI Root Bridge Controller that includes an instance of the **EFI_DEVICE_PATH_PROTOCOL** and the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Section 12.2 describes the PCI Root Bridge I/O Protocol in detail, and Section 12.2.1 describes how to build device paths for PCI Root Bridges. The **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** does not abstract access to the chipset-specific registers that are used to manage a PCI Root Bridge. This functionality is hidden within the system firmware or the EFI driver that produces the handles that represent the PCI Root Bridges.

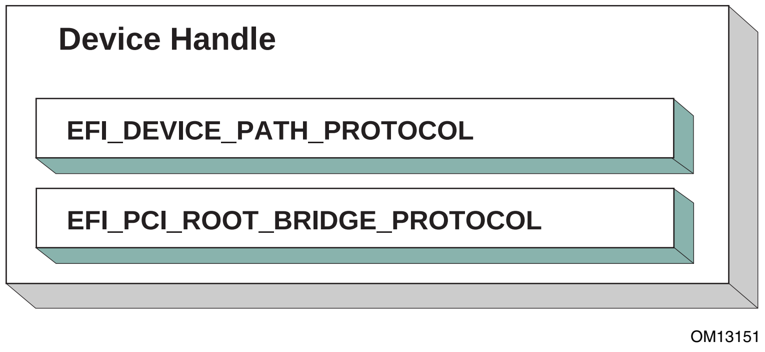


Figure 12-2. Device Handle for a PCI Root Bridge Controller

12.1.1.1 Sample PCI Architectures

The PCI Root Bridge I/O Protocol is designed to provide a software abstraction for a wide variety of PCI architectures including the ones described in this section. This section is not intended to be an exhaustive list of the PCI architectures that the PCI Root Bridge I/O Protocol can support. Instead, it is intended to show the flexibility of this protocol to adapt to current and future platform designs.

Figure 12-3 shows an example of a PCI Host Bus with one PCI Root Bridge. This PCI Root Bridge produces one PCI Local Bus that can contain PCI Devices on the motherboard and/or PCI slots. This would be typical of a desktop system. A higher end desktop system might contain a second PCI Root Bridge for AGP devices. The firmware for this platform would produce one instance of the PCI Root Bridge I/O Protocol.

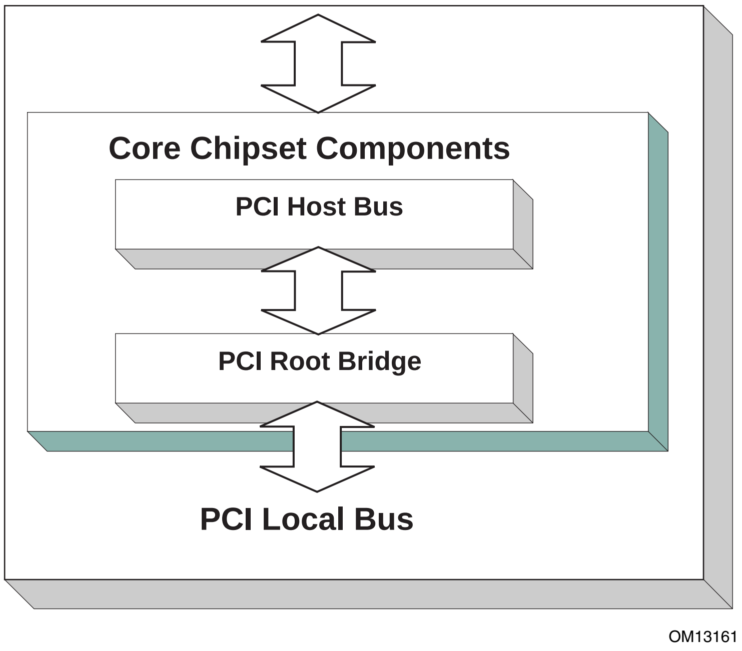


Figure 12-3. Desktop System with One PCI Root Bridge

Figure 12-4 shows an example of a larger server with one PCI Host Bus and four PCI Root Bridges. The PCI devices attached to the PCI Root Bridges are all part of the same coherency domain. This means they share a common PCI I/O Space, a common PCI Memory Space, and a common PCI Prefetchable Memory Space. Each PCI Root Bridge produces one PCI Local Bus that can contain PCI Devices on the motherboard or PCI slots. The firmware for this platform would produce four instances of the PCI Root Bridge I/O Protocol.

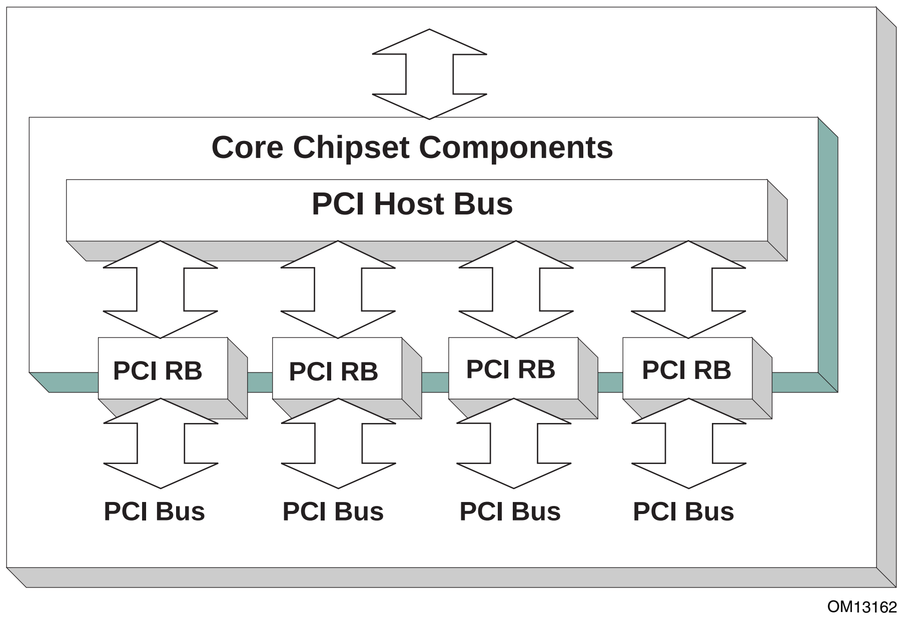


Figure 12-4. Server System with Four PCI Root Bridges

Figure 12-5 shows an example of a server with one PCI Host Bus and two PCI Root Bridges. Each of these PCI Root Bridges is a different PCI Segment which allows the system to have up to 512 PCI Buses. A single PCI Segment is limited to 256 PCI Buses. These two segments do not share the same PCI Configuration Space, but they do share the same PCI I/O, PCI Memory, and PCI Prefetchable Memory Space. This is why it can be described by a single PCI Host Bus. The firmware for this platform would produce two instances of the PCI Root Bridge I/O Protocol.

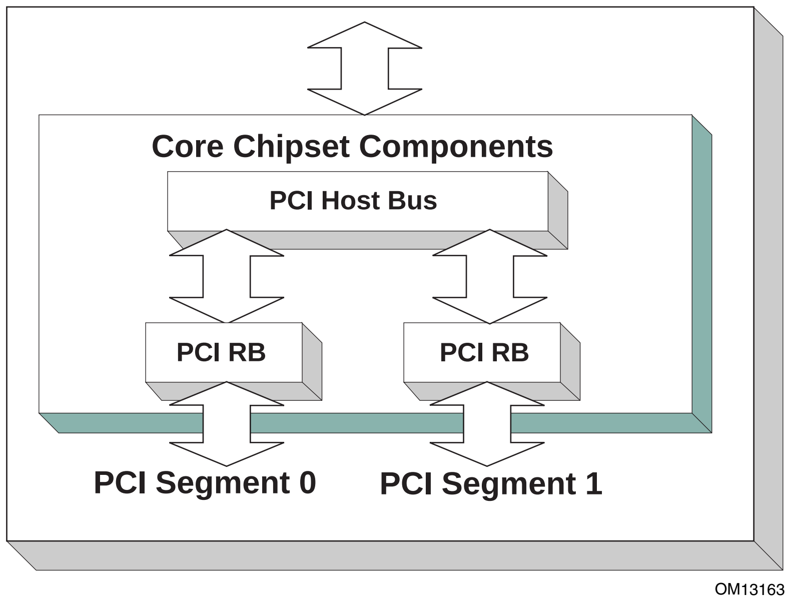


Figure 12-5. Server System with Two PCI Segments

Figure 12-6 shows a server system with two PCI Host Buses and one PCI Root Bridge per PCI Host Bus. This system supports up to 512 PCI Buses, but the PCI I/O, PCI Memory Space, and PCI Prefetchable Memory Space are not shared between the two PCI Root Bridges. The firmware for this platform would produce two instances of the PCI Root Bridge I/O Protocol.

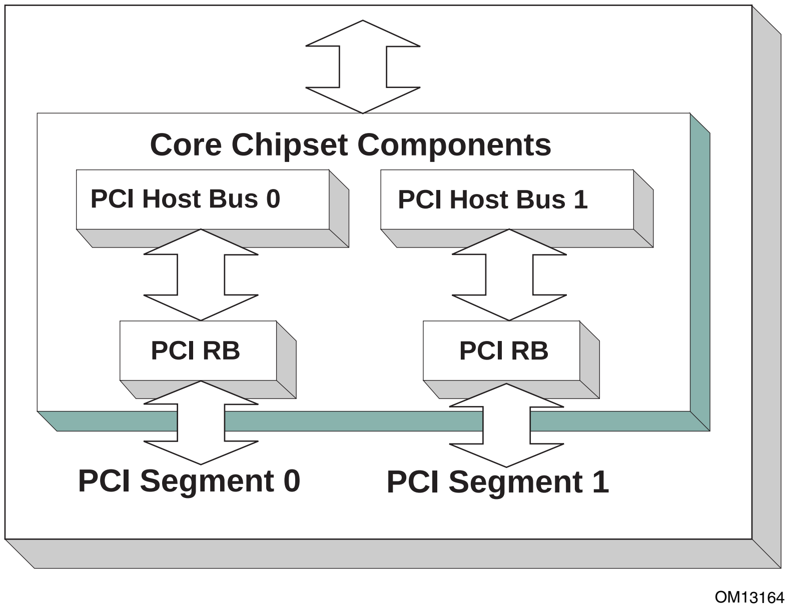


Figure 12-6. Server System with Two PCI Host Buses

12.2 PCI Root Bridge I/O Protocol

This section provides detailed information on the PCI Root Bridge I/O Protocol and its functions.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL

Summary

Provides the basic Memory, I/O, PCI configuration, and DMA interfaces that are used to abstract accesses to PCI controllers behind a PCI Root Bridge Controller.

GUID

```
#define EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_GUID  \
{0x2F707EBB,0x4A1A,0x11d4,0x9A,0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL {
  EFI_HANDLE                                   ParentHandle;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM     PollMem;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM     PollIo;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS          Mem;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS          Io;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS          Pci;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_COPY_MEM        CopyMem;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_MAP             Map;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_UNMAP           Unmap;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ALLOCATE_BUFFER AllocateBuffer;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FREE_BUFFER     FreeBuffer;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FLUSH           Flush;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_GET_ATTRIBUTES  GetAttributes;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_SET_ATTRIBUTES  SetAttributes;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_CONFIGURATION   Configuration;
  UINT32                                       SegmentNumber;
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL;
```

Parameters

ParentHandle	The **EFI_HANDLE** of the PCI Host Bridge of which this PCI Root Bridge is a member.
PollMem	Polls an address in memory mapped I/O space until an exit condition is met, or a timeout occurs. See the **PollMem()** function description.
PollIo	Polls an address in I/O space until an exit condition is met, or a timeout occurs. See the **PollIo()** function description.

Mem.Read	Allows reads from memory mapped I/O space. See the **Mem.Read()** function description.
Mem.Write	Allows writes to memory mapped I/O space. See the **Mem.Write()** function description.
Io.Read	Allows reads from I/O space. See the **Io.Read()** function description.
Io.Write	Allows writes to I/O space. See the **Io.Write()** function description.
Pci.Read	Allows reads from PCI configuration space. See the **Pci.Read()** function description.
Pci.Write	Allows writes to PCI configuration space. See the **Pci.Write()** function description.
CopyMem	Allows one region of PCI root bridge memory space to be copied to another region of PCI root bridge memory space. See the **CopyMem()** function description.
Map	Provides the PCI controller–specific addresses needed to access system memory for DMA. See the **Map()** function description.
Unmap	Releases any resources allocated by **Map()**. See the **Unmap()** function description.
AllocateBuffer	Allocates pages that are suitable for a common buffer mapping. See the **AllocateBuffer()** function description.
FreeBuffer	Free pages that were allocated with **AllocateBuffer()**. See the **FreeBuffer()** function description.
Flush	Flushes all PCI posted write transactions to system memory. See the **Flush()** function description.
GetAttributes	Gets the attributes that a PCI root bridge supports setting with **SetAttributes()**, and the attributes that a PCI root bridge is currently using. See the **GetAttributes()** function description.
SetAttributes	Sets attributes for a resource range on a PCI root bridge. See the **SetAttributes()** function description.
Configuration	Gets the current resource settings for this PCI root bridge. See the **Configuration()** function description.
SegmentNumber	The segment number that this PCI root bridge resides.

Related Definitions

```
//*******************************************************
// EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH
//*******************************************************
typedef enum {
  EfiPciWidthUint8,
  EfiPciWidthUint16,
  EfiPciWidthUint32,
  EfiPciWidthUint64,
  EfiPciWidthFifoUint8,
  EfiPciWidthFifoUint16,
  EfiPciWidthFifoUint32,
  EfiPciWidthFifoUint64,
  EfiPciWidthFillUint8,
  EfiPciWidthFillUint16,
  EfiPciWidthFillUint32,
  EfiPciWidthFillUint64,
  EfiPciWidthMaximum
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH;

//*******************************************************
// EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM
//*******************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM) (
  IN  struct EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL  *This,
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH   Width,
  IN  UINT64                                  Address,
  IN  UINT64                                  Mask,
  IN  UINT64                                  Value,
  IN  UINT64                                  Delay,
  OUT UINT64                                  *Result
  );

//*******************************************************
// EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM
//*******************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM) (
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL        *This,
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH  Width,
  IN     UINT64                                 Address,
  IN     UINTN                                  Count,
  IN OUT VOID                                   *Buffer
  );
```

```
//*******************************************************
// EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS
//*******************************************************
typedef struct {
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM  Read;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM  Write;
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS;

//*******************************************************
// EFI PCI Root Bridge I/O Protocol Attribute bits
//*******************************************************
#define EFI_PCI_ATTRIBUTE_ISA_MOTHERBOARD_IO    0x0001
#define EFI_PCI_ATTRIBUTE_ISA_IO                0x0002
#define EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO        0x0004
#define EFI_PCI_ATTRIBUTE_VGA_MEMORY            0x0008
#define EFI_PCI_ATTRIBUTE_VGA_IO                0x0010
#define EFI_PCI_ATTRIBUTE_IDE_PRIMARY_IO        0x0020
#define EFI_PCI_ATTRIBUTE_IDE_SECONDARY_IO      0x0040
#define EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE  0x0080
#define EFI_PCI_ATTRIBUTE_MEMORY_CACHED         0x0800
#define EFI_PCI_ATTRIBUTE_MEMORY_DISABLE        0x1000
#define EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE    0x8000
```

EFI_PCI_ATTRIBUTE_ISA_MOTHERBOARD_IO

If this bit is set, then the PCI I/O cycles between 0x00000000 and 0x000000FF are forwarded onto a PCI root bridge. This bit is used to forward I/O cycles for ISA motherboard devices onto a PCI root bridge.

EFI_PCI_ATTRIBUTE_ISA_IO

If this bit is set, then the PCI I/O cycles between 0x100 and 0x3FF are forwarded onto a PCI root bridge using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O cycles for legacy ISA devices onto a PCI root bridge.

EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO

If this bit is set, then the PCI I/O write cycles for 0x3C6, 0x3C8, and 0x3C9 are forwarded onto a PCI root bridge using a 10 bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O write cycles to the VGA palette registers onto a PCI root bridge.

EFI_PCI_ATTRIBUTE_VGA_MEMORY

If this bit is set, then the PCI memory cycles between 0xA0000 and 0xBFFFF are forwarded onto a PCI root bridge. This bit is used to forward memory cycles for a VGA frame buffer onto a PCI root bridge.

EFI_PCI_ATTRIBUTE_VGA_IO

If this bit is set, then the PCI I/O cycles in the ranges 0x3B0-0x3BB and 0x3C0-0x3DF are forwarded onto a PCI root bridge using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and the address bits 16..31 must be zero. This bit is used to forward I/O cycles for a VGA controller onto a PCI root bridge. Since **EFI_PCI_ATTRIBUTE_ENABLE_VGA_IO** also includes the I/O range described by **EFI_PCI_ATTRIBUTE_ENABLE_VGA_PALETTE_IO**, the **EFI_PCI_ATTRIBUTE_ENABLE_VGA_PALETTE_IO** bit is ignored if **EFI_PCI_ATTRIBUTE_ENABLE_VGA_IO** is set.

EFI_PCI_ATTRIBUTE_IDE_PRIMARY_IO

If this bit is set, then the PCI I/O cycles in the ranges 0x1F0-0x1F7 and 0x3F6-0x3F7 are forwarded onto a PCI root bridge using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O cycles for a Primary IDE controller onto a PCI root bridge.

EFI_PCI_ATTRIBUTE_IDE_SECONDARY_IO

If this bit is set, then the PCI I/O cycles in the ranges 0x170-0x177 and 0x376-0x377 are forwarded onto a PCI root bridge using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O cycles for a Secondary IDE controller onto a PCI root bridge.

EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE

If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is accessed in a write combining mode. By default, PCI memory ranges are not accessed in a write combining mode.

EFI_PCI_ATTRIBUTE_MEMORY_CACHED

If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is accessed in a cached mode. By default, PCI memory ranges are accessed noncached.

EFI_PCI_ATTRIBUTE_MEMORY_DISABLE

If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is disabled, and can no longer be accessed. By default, all PCI memory ranges are enabled.

EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE

This bit may only be used in the *Attributes* parameter to **AllocateBuffer()**. If this bit is set, then the PCI controller that is requesting a buffer through **AllocateBuffer()** is capable of producing PCI Dual Address Cycles, so it is able to access a 64-bit address space. If this bit is not set, then the PCI controller that is requesting a buffer through **AllocateBuffer()** is not capable of producing PCI Dual Address Cycles, so it is only able to access a 32-bit address space.

```
//*******************************************************
// EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION
//*******************************************************
typedef enum {
  EfiPciOperationBusMasterRead,
  EfiPciOperationBusMasterWrite,
  EfiPciOperationBusMasterCommonBuffer,
  EfiPciOperationBusMasterRead64,
  EfiPciOperationBusMasterWrite64,
  EfiPciOperationBusMasterCommonBuffer64,
  EfiPciOperationMaximum
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION;
```

`EfiPciOperationBusMasterRead`

A read operation from system memory by a bus master that is not capable of producing PCI dual address cycles.

`EfiPciOperationBusMasterWrite`

A write operation to system memory by a bus master that is not capable of producing PCI dual address cycles.

`EfiPciOperationBusMasterCommonBuffer`

Provides both read and write access to system memory by both the processor and a bus master that is not capable of producing PCI dual address cycles. The buffer is coherent from both the processor's and the bus master's point of view.

`EfiPciOperationBusMasterRead64`

A read operation from system memory by a bus master that is capable of producing PCI dual address cycles.

`EfiPciOperationBusMasterWrite64`

A write operation to system memory by a bus master that is capable of producing PCI dual address cycles.

`EfiPciOperationBusMasterCommonBuffer64`

Provides both read and write access to system memory by both the processor and a bus master that is capable of producing PCI dual address cycles. The buffer is coherent from both the processor's and the bus master's point of view.

Description

The **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** provides the basic Memory, I/O, PCI configuration, and DMA interfaces that are used to abstract accesses to PCI controllers. There is one **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** instance for each PCI root bridge in a system. Embedded systems, desktops, and workstations will typically only have one PCI root bridge. High-end servers may have multiple PCI root bridges. A device driver that wishes to manage a PCI bus in a system will have to retrieve the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** instance that is associated with the PCI bus to be managed. A device handle for a PCI Root Bridge will minimally contain an **EFI_DEVICE_PATH** instance and an **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** instance. The PCI bus driver can look at the **EFI_DEVICE_PATH** instances to determine which **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** instance to use.

Bus mastering PCI controllers can use the DMA services for DMA operations. There are three basic types of bus mastering DMA that is supported by this protocol. These are DMA reads by a bus master, DMA writes by a bus master, and common buffer DMA. The DMA read and write operations may need to be broken into smaller chunks. The caller of **Map()** must pay attention to the number of bytes that were mapped, and if required, loop until the entire buffer has been transferred. The following is a list of the different bus mastering DMA operations that are supported, and the sequence of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** APIs that are used for each DMA operation type. See "Related Definitions" above for the definition of the different DMA operation types.

DMA Bus Master Read Operation

- Call `Map()` for `EfiPciOperationBusMasterRead` or `EfiPciOperationBusMasterRead64`.
- Program the DMA Bus Master with the *`DeviceAddress`* returned by `Map()`.
- Start the DMA Bus Master.
- Wait for DMA Bus Master to complete the read operation.
- Call `Unmap()`.

DMA Bus Master Write Operation

- Call `Map()` for `EfiPciOperationBusMasterWrite` or `EfiPciOperationBusMasterRead64`.
- Program the DMA Bus Master with the *`DeviceAddress`* returned by `Map()`.
- Start the DMA Bus Master.
- Wait for DMA Bus Master to complete the write operation.
- Perform a PCI controller specific read transaction to flush all PCI write buffers (See *PCI Specification* Section 3.2.5.2) .
- Call `Flush()`.
- Call `Unmap()`.

DMA Bus Master Common Buffer Operation

- Call `AllocateBuffer()` to allocate a common buffer.
- Call `Map()` for `EfiPciOperationBusMasterCommonBuffer` or `EfiPciOperationBusMasterCommonBuffer64`.
- Program the DMA Bus Master with the *`DeviceAddress`* returned by `Map()`.
- The common buffer can now be accessed equally by the processor and the DMA bus master.
- Call `Unmap()`.
- Call `FreeBuffer()`.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.PollMem()

Summary

Reads from the memory space of a PCI Root Bridge. Returns when either the polling exit criteria is satisfied or after a defined duration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM) (
  IN  struct EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *This,
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH    Width,
  IN  UINT64                                   Address,
  IN  UINT64                                   Mask,
  IN  UINT64                                   Value,
  IN  UINT64                                   Delay,
  OUT UINT64                                   *Result
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Width	Signifies the width of the memory operations. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH** is defined in Section 12.2.
Address	The base address of the memory operations. The caller is responsible for aligning *Address* if required.
Mask	Mask used for the polling criteria. Bytes above *Width* in *Mask* are ignored. The bits in the bytes below *Width* which are zero in *Mask* are ignored when polling the memory address.
Value	The comparison value used for the polling exit criteria.
Delay	The number of 100 ns units to poll. Note that timer available may be of poorer granularity.
Result	Pointer to the last value read from the memory location.

Description

This function provides a standard way to poll a PCI memory location. A PCI memory read operation is performed at the PCI memory address specified by *Address* for the width specified by *Width*. The result of this PCI memory read operation is stored in *Result*. This PCI memory read operation is repeated until either a timeout of *Delay* 100 ns units has expired, or (*Result* & *Mask)* is equal to *Value*.

This function will always perform at least one PCI memory read access no matter how small *Delay* may be. If *Delay* is zero, then *Result* will be returned with a status of **EFI_SUCCESS** even if *Result* does not match the exit criteria. If *Delay* expires, then **EFI_TIMEOUT** is returned.

If *Width* is not **EfiPciWidthUint8**, **EfiPciWidthUint16**, **EfiPciWidthUint32**, or **EfiPciWidthUint64**, then **EFI_INVALID_PARAMETER** is returned.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI Root Bridge on a platform might require. For example on some platforms, width requests of **EfiPciWidthUint64** are not supported.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. However, if the memory mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Status Codes Returned

EFI_SUCCESS	The last data returned from the access matched the poll exit criteria.
EFI_INVALID_PARAMETER	*Width* is invalid.
EFI_INVALID_PARAMETER	*Result* is **NULL**.
EFI_TIMEOUT	*Delay* expired before a match occurred.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.PollIo()

Summary

Reads from the I/O space of a PCI Root Bridge. Returns when either the polling exit criteria is satisfied or after a defined duration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM) (
  IN  struct EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *This,
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH    Width,
  IN  UINT64                                   Address,
  IN  UINT64                                   Mask,
  IN  UINT64                                   Value,
  IN  UINT64                                   Delay,
  OUT UINT64                                   *Result
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Width	Signifies the width of the I/O operations. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH** is defined in Section 12.2.
Address	The base address of the I/O operations. The caller is responsible for aligning *Address* if required.
Mask	Mask used for the polling criteria. Bytes above *Width* in *Mask* are ignored. The bits in the bytes below *Width* which are zero in *Mask* are ignored when polling the I/O address.
Value	The comparison value used for the polling exit criteria.
Delay	The number of 100 ns units to poll. Note that timer available may be of poorer granularity.
Result	Pointer to the last value read from the memory location.

Description

This function provides a standard way to poll a PCI I/O location. A PCI I/O read operation is performed at the PCI I/O address specified by *Address* for the width specified by *Width*. The result of this PCI I/O read operation is stored in *Result*. This PCI I/O read operation is repeated until either a timeout of *Delay* 100 ns units has expired, or (*Result* & *Mask)* is equal to *Value*.

This function will always perform at least one I/O access no matter how small *Delay* may be. If *Delay* is zero, then *Result* will be returned with a status of **EFI_SUCCESS** even if *Result* does not match the exit criteria. If *Delay* expires, then **EFI_TIMEOUT** is returned.

If *Width* is not **EfiPciWidthUint8**, **EfiPciWidthUint16**, **EfiPciWidthUint32**, or **EfiPciWidthUint64**, then **EFI_INVALID_PARAMETER** is returned.

The I/O operations are carried out exactly as requested. The caller is responsible satisfying any alignment and I/O width restrictions that the PCI Root Bridge on a platform might require. For example on some platforms, width requests of **EfiPciWidthUint64** do not work.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Status Codes Returned

EFI_SUCCESS	The last data returned from the access matched the poll exit criteria.
EFI_INVALID_PARAMETER	*Width* is invalid.
EFI_INVALID_PARAMETER	*Result* is **NULL**.
EFI_TIMEOUT	*Delay* expired before a match occurred.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Mem.Read()
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Mem.Write()

Summary

Enables a PCI driver to access PCI controller registers in the PCI root bridge memory space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM) (
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL        *This,
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH  Width,
  IN     UINT64                                 Address,
  IN     UINTN                                  Count,
  IN OUT VOID                                   *Buffer
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Width	Signifies the width of the memory operation. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH** is defined in Section 12.2.
Address	The base address of the memory operation. The caller is responsible for aligning the *Address* if required.
Count	The number of memory operations to perform. Bytes moved is *Width* size * *Count*, starting at *Address*.
Buffer	For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description

The **Mem.Read()**, and **Mem.Write()** functions enable a driver to access PCI controller registers in the PCI root bridge memory space.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI Root Bridge on a platform might require. For example on some platforms, width requests of **EfiPciWidthUint64** do not work.

If *Width* is **EfiPciWidthUint8**, **EfiPciWidthUint16**, **EfiPciWidthUint32**, or **EfiPciWidthUint64**, then both *Address* and *Buffer* are incremented for each of the *Count* operations performed.

If *Width* is **EfiPciWidthFifoUint8**, **EfiPciWidthFifoUint16**, **EfiPciWidthFifoUint32**, or **EfiPciWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiPciWidthFillUint8**, **EfiPciWidthFillUint16**, **EfiPciWidthFillUint32**, or **EfiPciWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

All the PCI read transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the *PCI Specification*. However, if the memory-mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PCI root bridge.
EFI_INVALID_PARAMETER	*Width* is invalid for this PCI root bridge.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Io.Read()
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Io.Write()

Summary

Enables a PCI driver to access PCI controller registers in the PCI root bridge I/O space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM) (
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL         *This,
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH   Width,
  IN     UINT64                                  Address,
  IN     UINTN                                   Count,
  IN OUT VOID                                    *Buffer
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Width	Signifies the width of the memory operations. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH** is defined in Section 12.2.
Address	The base address of the I/O operation. The caller is responsible for aligning the *Address* if required.
Count	The number of I/O operations to perform. Bytes moved is *Width* size * *Count*, starting at *Address*.
Buffer	For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description

The **`Io.Read()`**, and **`Io.Write()`** functions enable a driver to access PCI controller registers in the PCI root bridge I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and I/O width restrictions that a PCI root bridge on a platform might require. For example on some platforms, width requests of **`EfiPciWidthUint64`** do not work.

If *`Width`* is **`EfiPciWidthUint8`**, **`EfiPciWidthUint16`**, **`EfiPciWidthUint32`**, or **`EfiPciWidthUint64`**, then both *`Address`* and *`Buffer`* are incremented for each of the *`Count`* operations performed.

If *`Width`* is **`EfiPciWidthFifoUint8`**, **`EfiPciWidthFifoUint16`**, **`EfiPciWidthFifoUint32`**, or **`EfiPciWidthFifoUint64`**, then only *`Buffer`* is incremented for each of the *`Count`* operations performed. The read or write operation is performed *`Count`* times on the same *`Address`*.

If *`Width`* is **`EfiPciWidthFillUint8`**, **`EfiPciWidthFillUint16`**, **`EfiPciWidthFillUint32`**, or **`EfiPciWidthFillUint64`**, then only *`Address`* is incremented for each of the *`Count`* operations performed. The read or write operation is performed *`Count`* times from the first element of *`Buffer`*.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PCI root bridge.
EFI_INVALID_PARAMETER	*`Width`* is invalid for this PCI root bridge.
EFI_INVALID_PARAMETER	*`Buffer`* is **`NULL`**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Read() EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Write()

Summary

Enables a PCI driver to access PCI controller registers in a PCI root bridge's configuration space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM) (
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL        *This,
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH  Width,
  IN     UINT64                                 Address,
  IN     UINTN                                  Count,
  IN OUT VOID                                   *Buffer
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Width	Signifies the width of the memory operations. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH** is defined in Section 12.2.
Address	The address within the PCI configuration space for the PCI controller. See Table 12-1 for the format of *Address*.
Count	The number of PCI configuration operations to perform. Bytes moved is *Width* size * *Count*, starting at *Address*.
Buffer	For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description

The **Pci.Read()** and **Pci.Write()** functions enable a driver to access PCI configuration registers for a PCI controller.

The PCI Configuration operations are carried out exactly as requested. The caller is responsible for any alignment and PCI configuration width issues that a PCI Root Bridge on a platform might require. For example on some platforms, width requests of **EfiPciWidthUint64** do not work.

If *Width* is **EfiPciWidthUint8**, **EfiPciWidthUint16**, **EfiPciWidthUint32**, or **EfiPciWidthUint64**, then both *Address* and *Buffer* are incremented for each of the *Count* operations performed.

If *Width* is **EfiPciWidthFifoUint8**, **EfiPciWidthFifoUint16**, **EfiPciWidthFifoUint32**, or **EfiPciWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiPciWidthFillUint8**, **EfiPciWidthFillUint16**, **EfiPciWidthFillUint32**, or **EfiPciWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Table 12-1. PCI Configuration Address

Mnemonic	Byte Offset	Byte Length	Description
Register	0	1	The register number on the PCI Function.
Function	1	1	The PCI Function number on the PCI Device.
Device	2	1	The PCI Device number on the PCI Bus.
Bus	3	1	The PCI Bus number.
ExtendedRegister	4	4	The register number on the PCI Function. If this field is zero, then the Register field is used for the register number. If this field is nonzero, then the Register field is ignored, and the ExtendedRegister field is used for the register number.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PCI root bridge.
EFI_INVALID_PARAMETER	*Width* is invalid for this PCI root bridge.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.CopyMem()

Summary

Enables a PCI driver to copy one region of PCI root bridge memory space to another region of PCI root bridge memory space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_COPY_MEM) (
  IN    EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL        *This,
  IN    EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH  Width,
  IN    UINT64                                 DestAddress,
  IN    UINT64                                 SrcAddress,
  IN    UINTN                                  Count
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** instance. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Width	Signifies the width of the memory operations. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH** is defined in Section 12.2.
DestAddress	The destination address of the memory operation. The caller is responsible for aligning the *DestAddress* if required.
SrcAddress	The source address of the memory operation. The caller is responsible for aligning the *SrcAddress* if required.
Count	The number of memory operations to perform. Bytes moved is *Width* size * *Count*, starting at *DestAddress* and *SrcAddress*.

Description

The **CopyMem()** function enables a PCI driver to copy one region of PCI root bridge memory space to another region of PCI root bridge memory space. This is especially useful for video scroll operation on a memory mapped video buffer.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI root bridge on a platform might require. For example on some platforms, width requests of **EfiPciWidthUint64** do not work.

If *Width* is **EfiPciWidthUint8**, **EfiPciWidthUint16**, **EfiPciWidthUint32**, or **EfiPciWidthUint64**, then *Count* read/write transactions are performed to move the contents of the *SrcAddress* buffer to the *DestAddress* buffer. The implementation must be reentrant, and it must handle overlapping *SrcAddress* and *DestAddress* buffers. This means that the implementation of **CopyMem()** must choose the correct direction of the copy operation based on the type of overlap that exists between the *SrcAddress* and *DestAddress* buffers. If either the *SrcAddress* buffer or the *DestAddress* buffer crosses the top of the processor's address space, then the result of the copy operation is unpredictable.

The contents of the *DestAddress* buffer on exit from this service must match the contents of the *SrcAddress* buffer on entry to this service. Due to potential overlaps, the contents of the *SrcAddress* buffer may be modified by this service. The following rules can be used to guarantee the correct behavior:

1. If *DestAddress* > *SrcAddress* **and** *DestAddress* < (*SrcAddress* + *Width* size * *Count*), then the data should be copied from the *SrcAddress* buffer to the *DestAddress* buffer starting from the end of buffers and working toward the beginning of the buffers.
2. Otherwise, the data should be copied from the *SrcAddress* buffer to the *DestAddress* buffer starting from the beginning of the buffers and working toward the end of the buffers.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the *PCI Specification*. However, if the memory-mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Status Codes Returned

EFI_SUCCESS	The data was copied from one memory region to another memory region.
EFI_INVALID_PARAMETER	*Width* is invalid for this PCI root bridge.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Map()

Summary

Provides the PCI controller–specific addresses required to access system memory from a DMA bus master.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_MAP) (
  IN      EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL           *This,
  IN      EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION Operation,
  IN      VOID                                      *HostAddress,
  IN OUT  UINTN                                     *NumberOfBytes,
  OUT     EFI_PHYSICAL_ADDRESS                      *DeviceAddress,
  OUT     VOID                                      **Mapping
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Operation	Indicates if the bus master is going to read or write to system memory. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION** is defined in Section 12.2.
HostAddress	The system memory address to map to the PCI controller.
NumberOfBytes	On input the number of bytes to map. On output the number of bytes that were mapped.
DeviceAddress	The resulting map address for the bus master PCI controller to use to access the system memory's *HostAddress*. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5. This address cannot be used by the processor to access the contents of the buffer specified by *HostAddress*.
Mapping	The value to pass to **Unmap()** when the bus master DMA operation is complete.

Description

The **Map()** function provides the PCI controller specific addresses needed to access system memory. This function is used to map system memory for PCI bus master DMA accesses.

All PCI bus master accesses must be performed through their mapped addresses and such mappings must be freed with **Unmap()** when complete. If the bus master access is a single read or single write data transfer, then **EfiPciOperationBusMasterRead**, **EfiPciOperationBusMasterRead64**, **EfiPciOperationBusMasterWrite**, or **EfiPciOperationBusMasterWrite64** is used and the range is unmapped to complete the operation. If performing an **EfiPciOperationBusMasterRead** or **EfiPciOperationBusMasterRead64** operation, all the data must be present in system memory before **Map()** is performed. Similarly, if performing an **EfiPciOperation-BusMasterWrite** or **EfiPciOperationBusMasterWrite64** the data cannot be properly accessed in system memory until **Unmap()** is performed.

Bus master operations that require both read and write access or require multiple host device interactions within the same mapped region must use **EfiPciOperation-BusMasterCommonBuffer** or **EfiPciOperationBusMasterCommonBuffer64**. However, only memory allocated via the **AllocateBuffer()** interface can be mapped for this type of operation.

In all mapping requests the resulting *NumberOfBytes* actually mapped may be less than the requested amount. In this case, the DMA operation will have to be broken up into smaller chunks. The **Map()** function will map as much of the DMA operation as it can at one time. The caller may have to loop on **Map()** and **Unmap()** in order to complete a large DMA transfer.

Status Codes Returned

EFI_SUCCESS	The range was mapped for the returned *NumberOfBytes*.
EFI_INVALID_PARAMETER	*Operation* is invalid.
EFI_INVALID_PARAMETER	*HostAddress* is **NULL**.
EFI_INVALID_PARAMETER	*NumberOfBytes* is **NULL**.
EFI_INVALID_PARAMETER	*DeviceAddress* is **NULL**.
EFI_INVALID_PARAMETER	*Mapping* is **NULL**.
EFI_UNSUPPORTED	The *HostAddress* cannot be mapped as a common buffer.
EFI_DEVICE_ERROR	The system hardware could not map the requested address.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Unmap()

Summary

Completes the **Map()** operation and releases any corresponding resources.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_UNMAP) (
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *This,
  IN  VOID                              *Mapping
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Mapping	The mapping value returned from **Map()**.

Description

The **Unmap()** function completes the **Map()** operation and releases any corresponding resources. If the operation was an **EfiPciOperationBusMasterWrite** or **EfiPciOperationBusMasterWrite64**, the data is committed to the target system memory. Any resources used for the mapping are freed.

Status Codes Returned

EFI_SUCCESS	The range was unmapped.
EFI_INVALID_PARAMETER	*Mapping* is not a value that was returned by **Map()**.
EFI_DEVICE_ERROR	The data was not committed to the target system memory.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.AllocateBuffer()

Summary

Allocates pages that are suitable for an **EfiPciOperationBusMasterCommonBuffer** or **EfiPciOperationBusMasterCommonBuffer64** mapping.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ALLOCATE_BUFFER) (
  IN      EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *This,
  IN      EFI_ALLOCATE_TYPE                 Type,
  IN      EFI_MEMORY_TYPE                   MemoryType,
  IN      UINTN                             Pages,
  OUT     VOID                              **HostAddress,
  IN      UINT64                            Attributes
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.1.
Type	This parameter is not used and must be ignored.
MemoryType	The type of memory to allocate, **EfiBootServicesData** or **EfiRuntimeServicesData**. Type **EFI_MEMORY_TYPE** is defined in Chapter 5.
Pages	The number of pages to allocate.
HostAddress	A pointer to store the base system memory address of the allocated range.

Attributes	The requested bit mask of attributes for the allocated range. Only the attributes **EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE**, **EFI_PCI_ATTRIBUTE_MEMORY_CACHED**, and **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE** may be used with this function. If any other bits are set, then **EFI_UNSUPPORTED** is returned. This function may choose to ignore this bit mask. The **EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE**, and **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attributes provide a hint to the implementation that may improve the performance of the calling driver. The implementation may choose any default for the memory attributes including write combining, cached, both, or neither as long as the allocated buffer can be seen equally by both the processor and the PCI bus master.

Description

The **AllocateBuffer()** function allocates pages that are suitable for an **EfiPciOperationBusMasterCommonBuffer** or **EfiPciOperationBusMasterCommonBuffer64** mapping. This means that the buffer allocated by this function must support simultaneous access by both the processor and a PCI Bus Master. The device address that the PCI Bus Master uses to access the buffer can be retrieved with a call to **Map()**.

If the **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE** bit of *Attributes* is set, then when the buffer allocated by this function is mapped with a call to **Map()**, the device address that is returned by **Map()** must be within the 64-bit device address space of the PCI Bus Master.

If the **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE** bit of *Attributes* is clear, then when the buffer allocated by this function is mapped with a call to **Map()**, the device address that is returned by **Map()** must be within the 32-bit device address space of the PCI Bus Master.

If the memory allocation specified by *MemoryType* and *Pages* cannot be satisfied, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were allocated.
EFI_INVALID_PARAMETER	*MemoryType* is invalid.
EFI_INVALID_PARAMETER	*HostAddress* is **NULL**.
EFI_UNSUPPORTED	*Attributes* is unsupported. The only legal attribute bits are **MEMORY_WRITE_COMBINE**, **MEMORY_CACHED**, and **DUAL_ADDRESS_CYCLE**.
EFI_OUT_OF_RESOURCES	The memory pages could not be allocated.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.FreeBuffer()

Summary

Frees memory that was allocated with **AllocateBuffer()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FREE_BUFFER) (
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *This,
  IN  UINTN                             Pages,
  IN  VOID                              *HostAddress
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Pages	The number of pages to free.
HostAddress	The base system memory address of the allocated range.

Description

The **FreeBuffer()** function frees memory that was allocated with **AllocateBuffer()**.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were freed.
EFI_INVALID_PARAMETER	The memory range specified by *HostAddress* and *Pages* was not allocated with **AllocateBuffer()**.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Flush()

Summary

Flushes all PCI posted write transactions from a PCI host bridge to system memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FLUSH) (
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *This
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.1.

Description

The **Flush()** function flushes any PCI posted write transactions from a PCI host bridge to system memory. Posted write transactions are generated by PCI bus masters when they perform write transactions to target addresses in system memory.

This function does not flush posted write transactions from any PCI bridges. A PCI controller specific action must be taken to guarantee that the posted write transactions have been flushed from the PCI controller and from all the PCI bridges into the PCI host bridge. This is typically done with a PCI read transaction from the PCI controller prior to calling **Flush()**.

If the PCI controller specific action required to flush the PCI posted write transactions has been performed, and this function returns **EFI_SUCCESS**, then the PCI bus master's view and the processor's view of system memory are guaranteed to be coherent. If the PCI posted write transactions cannot be flushed from the PCI host bridge, then the PCI bus master and processor are not guaranteed to have a coherent view of system memory, and **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	The PCI posted write transactions were flushed from the PCI host bridge to system memory.
EFI_DEVICE_ERROR	The PCI posted write transactions were not flushed from the PCI host bridge due to a hardware error.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.GetAttributes()

Summary

Gets the attributes that a PCI root bridge supports setting with **SetAttributes()**, and the attributes that a PCI root bridge is currently using.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_GET_ATTRIBUTES) (
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *This,
  OUT UINT64                            *Supports    OPTIONAL,
  OUT UINT64                            *Attributes  OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Supports	A pointer to the mask of attributes that this PCI root bridge supports setting with **SetAttributes()**. The available attributes are listed in Section 12.2. This is an optional parameter that may be **NULL**.
Attributes	A pointer to the mask of attributes that this PCI root bridge is currently using. The available attributes are listed in Section 12.2. This is an optional parameter that may be **NULL**.

Description

The **GetAttributes()** function returns the mask of attributes that this PCI root bridge supports and the mask of attributes that the PCI root bridge is currently using. If *Supports* is not **NULL**, then *Supports* is set to the mask of attributes that the PCI root bridge supports. If *Attributes* is not **NULL**, then *Attributes* is set to the mask of attributes that the PCI root bridge is currently using. If both *Supports* and *Attributes* are **NULL**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, **EFI_SUCCESS** is returned.

If a bit is set in *Supports*, then the PCI root bridge supports this attribute type, and a call can be made to **SetAttributes()** using that attribute type. If a bit is set in *Attributes*, then the PCI root bridge is currently using that attribute type. Since a PCI host bus may be composed of more than one PCI root bridge, different *Attributes* values may be returned by different PCI root bridges.

Status Codes Returned

EFI_SUCCESS	If *Supports* is not **NULL**, then the attributes that the PCI root bridge supports is returned in *Supports*. If *Attributes* is not **NULL**, then the attributes that the PCI root bridge is currently using is returned in *Attributes*.
EFI_INVALID_PARAMETER	Both *Supports* and *Attributes* are **NULL**.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.SetAttributes()

Summary

Sets attributes for a resource range on a PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_SET_ATTRIBUTES) (
  IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL  *This,
  IN UINT64                           Attributes,
  IN OUT UINT64                       *ResourceBase    OPTIONAL,
  IN OUT UINT64                       *ResourceLength  OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Attributes	The mask of attributes to set. If the attribute bit **MEMORY_WRITE_COMBINE**, **MEMORY_CACHED**, or **MEMORY_DISABLE** is set, then the resource range is specified by *ResourceBase* and *ResourceLength*. If **MEMORY_WRITE_COMBINE**, **MEMORY_CACHED**, and **MEMORY_DISABLE** are not set, then *ResourceBase* and *ResourceLength* are ignored, and may be **NULL**. The available attributes are listed in Section 12.2.
ResourceBase	A pointer to the base address of the resource range to be modified by the attributes specified by *Attributes*. On return, **ResourceBase* will be set the actual base address of the resource range. Not all resources can be set to a byte boundary, so the actual base address may differ from the one passed in by the caller. This parameter is only used if the **MEMORY_WRITE_COMBINE** bit, the **MEMORY_CACHED** bit, or the **MEMORY_DISABLE** bit of *Attributes* is set. Otherwise, it is ignored, and may be **NULL**.

ResourceLength	A pointer to the length of the resource range to be modified by the attributes specified by *Attributes*. On return, **ResourceLength* will be set the actual length of the resource range. Not all resources can be set to a byte boundary, so the actual length may differ from the one passed in by the caller. This parameter is only used if the **MEMORY_WRITE_COMBINE** bit, the **MEMORY_CACHED** bit, or the **MEMORY_DISABLE** bit of *Attributes* is set. Otherwise, it is ignored, and may be **NULL**.

Description

The **SetAttributes()** function sets the attributes specified in *Attributes* for the PCI root bridge on the resource range specified by *ResourceBase* and *ResourceLength*. Since the granularity of setting these attributes may vary from resource type to resource type, and from platform to platform, the actual resource range and the one passed in by the caller may differ. As a result, this function may set the attributes specified by *Attributes* on a larger resource range than the caller requested. The actual range is returned in *ResourceBase* and *ResourceLength*. The caller is responsible for verifying that the actual range for which the attributes were set is acceptable.

If the attributes are set on the PCI root bridge, then the actual resource range is returned in *ResourceBase* and *ResourceLength*, and **EFI_SUCCESS** is returned.

If the attributes specified by *Attributes* are not supported by the PCI root bridge, then **EFI_UNSUPPORTED** is returned. The set of supported attributes for a PCI root bridge can be found by calling **GetAttributes()**.

If either *ResourceBase* or *ResourceLength* are **NULL**, and a resource range is required for the attributes specified in *Attributes*, then **EFI_INVALID_PARAMETER** is returned.

If more than one resource range is required for the set of attributes specified by *Attributes*, then **EFI_INVALID_PARAMETER** is returned.

If there are not enough resources available to set the attributes, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The set of attributes specified by *Attributes* for the resource range specified by *ResourceBase* and *ResourceLength* were set on the PCI root bridge, and the actual resource range is returned in *ResuourceBase* and *ResourceLength*.
EFI_UNSUPPORTED	A bit is set in *Attributes* that is not supported by the PCI Root Bridge. The supported attribute bits are reported by **GetAttributes()**.
EFI_INVALID_PARAMETER	More than one attribute bit is set in *Attributes* that requires a resource range.
EFI_INVALID_PARAMETER	A resource range is required, and *ResourceBase* is **NULL**.
EFI_INVALID_PARAMETER	A resource range is required, and *ResourceLength* is **NULL**.
EFI_OUT_OF_RESOURCES	There are not enough resources to set the attributes on the resource range specified by *BaseAddress* and *Length*.

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()

Summary

Retrieves the current resource settings of this PCI root bridge in the form of a set of ACPI 2.0 resource descriptors.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_CONFIGURATION) (
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *This,
  OUT VOID                              **Resources
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.
Resources	A pointer to the ACPI 2.0 resource descriptors that describe the current configuration of this PCI root bridge. The storage for the ACPI 2.0 resource descriptors is allocated by this function. The caller must treat the return buffer as read-only data, and the buffer must not be freed by the caller. See "Related Definitions" for the ACPI 2.0 resource descriptors that may be used.

Related Definitions

There are only two resource descriptor types from the *ACPI Specification* that may be used to describe the current resources allocated to a PCI root bridge. These are the QWORD Address Space Descriptor (ACPI 2.0 Section 6.4.3.5.1), and the End Tag (ACPI 2.0 Section 6.4.2.8). The QWORD Address Space Descriptor can describe memory, I/O, and bus number ranges for dynamic or fixed resources. The configuration of a PCI root bridge is described with one or more QWORD Address Space Descriptors followed by an End Tag. Table 12-2 and Table 12-3 contains these two descriptor types. Please see the *ACPI Specification* for details on the field values.

Table 12-2. ACPI 2.0 QWORD Address Space Descriptor

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x8A	QWORD Address Space Descriptor
0x01	0x02	0x2B	Length of this descriptor in bytes not including the first two fields
0x03	0x01		Resource Type 0 – Memory Range 1 – I/O Range 2 – Bus Number Range
0x04	0x01		General Flags
0x05	0x01		Type Specific Flags
0x06	0x08		Address Space Granularity
0x0E	0x08		Address Range Minimum
0x16	0x08		Address Range Maximum
0x1E	0x08		Address Translation Offset
0x26	0x08		Address Length

Table 12-3. ACPI 2.0 End Tag

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x79	End Tag
0x01	0x01	0x00	Checksum. If 0, then checksum is assumed to be valid.

Description

The **Configuration()** function retrieves a set of ACPI 2.0 resource descriptors that contains the current configuration of this PCI root bridge. If the current configuration can be retrieved, then it is returned in *Resources* and **EFI_SUCCESS** is returned. See "Related Definitions" below for the resource descriptor types that are supported by this function. If the current configuration cannot be retrieved, then **EFI_UNSUPPORTED** is returned.

Status Codes Returned

EFI_SUCCESS	The current configuration of this PCI root bridge was returned in *Resources*.
EFI_UNSUPPORTED	The current configuration of this PCI root bridge could not be retrieved.

12.2.1 PCI Root Bridge Device Paths

An **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** must be installed on a handle for its services to be available to EFI drivers. In addition to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**, an **EFI_DEVICE_PATH** must also be installed on the same handle. See Chapter 5 for a detailed description of the **EFI_DEVICE_PATH**.

Typically, an ACPI Device Path Node is used to describe a PCI Root Bridge. Depending on the bus hierarchy in the system, additional device path nodes may precede this ACPI Device Path Node. A desktop system will typically contain only one PCI Root Bridge, so there would be one handle with a **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** and an **EFI_DEVICE_PATH**. A server system may contain multiple PCI Root Bridges, so it would contain a handle for each PCI Root Bridge present, and on each of those handles would be an **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** and an **EFI_DEVICE_PATH**. In all cases, the contents of the ACPI Device Path Nodes for PCI Root Bridges must match the information present in the ACPI tables for that system.

Table 12-4 shows an example device path for a PCI Root Bridge in a desktop system. Today, a desktop system typically contains one PCI Root Bridge. This device path consists of an ACPI Device Path Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. For a system with only one PCI Root Bridge, the _UID value is usually 0x0000. The shorthand notation for this device path is **ACPI(PNP0A03,0)**.

Table 12-4. PCI Root Bridge Device Path for a Desktop System

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes
0x08	0x04	0x0000	_UID
0x0C	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x0D	0x01	0xFF	Sub type – End of Entire Device Path
0x0E	0x02	0x04	Length – 0x04 bytes

Table 12-5 through Table 12-8 show example device paths for the PCI Root Bridges in a server system with four PCI Root Bridges. Each of these device paths consists of an ACPI Device Path Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridges. The only difference between each of these device paths is the _UID field. The shorthand notation for these four device paths is `ACPI(PNP0A03,0)`, `ACPI(PNP0A03,1)`, `ACPI(PNP0A03,2)`, and `ACPI(PNP0A03,3)`.

Table 12-5. PCI Root Bridge Device Path for Bridge #0 in a Server System

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes
0x08	0x04	0x0000	_UID
0x0C	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x0D	0x01	0xFF	Sub type – End of Entire Device Path
0x0E	0x02	0x04	Length – 0x04 bytes

Table 12-6. PCI Root Bridge Device Path for Bridge #1 in a Server System

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes
0x08	0x04	0x0001	_UID
0x0C	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x0D	0x01	0xFF	Sub type – End of Entire Device Path
0x0E	0x02	0x04	Length – 0x04 bytes

Table 12-7. PCI Root Bridge Device Path for Bridge #2 in a Server System

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes
0x08	0x04	0x0002	_UID
0x0C	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x0D	0x01	0xFF	Sub type – End of Entire Device Path
0x0E	0x02	0x04	Length – 0x04 bytes

Table 12-8. PCI Root Bridge Device Path for Bridge #3 in a Server System

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x08	0x04	0x0003	_UID
0x0C	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x0D	0x01	0xFF	Sub type – End of Entire Device Path
0x0E	0x02	0x04	Length – 0x04 bytes

Table 12-9 shows an example device path for a PCI Root Bridge using an Expanded ACPI Device Path. This device path consists of an Expanded ACPI Device Path Node, and a Device Path End Structure. The _UID and _CID fields must match the ACPI table description of the PCI Root Bridge. For a system with only one PCI Root Bridge, the _UID value is usually 0x0000. The shorthand notation for this device path is `ACPI(12345678,0,PNP0A03)`.

Table 12-9. PCI Root Bridge Device Path Using Expanded ACPI Device Path

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x02	Sub type – Expanded ACPI Device Path
0x02	0x02	0x10	Length – 0x10 bytes
0x04	0x04	**0x1234, 0x5678**	_HID-device specific
0x08	0x04	0x0000	_UID
0x0C	**0x04**	0x41D0, 0x0A03	_CID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x10	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x11	0x01	0xFF	Sub type – End of Entire Device Path
0x12	0x02	0x04	Length – 0x04 bytes

12.3 PCI Driver Model

These sections (Sections 12.3 and 12.4) describe the PCI Driver Model. This includes the behavior of PCI Bus Drivers, the behavior of a PCI Device Drivers, and a detailed description of the PCI I/O Protocol. The PCI Bus Driver manages PCI buses present in a system, and PCI Device Drivers manage PCI controllers present on PCI buses. The PCI Device Drivers produce an I/O abstraction that can be used to boot an EFI compliant operating system.

This document provides enough material to implement a PCI Bus Driver, and the tools required to design and implement a PCI Device Drivers. It does not provide any information on specific PCI devices.

The material contained in this document is designed to extend the *EFI Specification* and the *EFI Driver Model Specification* in a way that supports PCI device drivers and PCI bus drivers. These extensions are provided in the form of PCI-specific protocols. This document provides the information required to implement a PCI Bus Driver in system firmware. The document also contains the information required by driver writers to design and implement PCI Device Drivers that a platform may need to boot an EFI compliant OS.

A full understanding of the *EFI Specification*, the *EFI Driver Model Specification*, and the *PCI Specification* is assumed throughout this document. The PCI Driver Model described here is intended to be a foundation on which a PCI Bus Driver and a wide variety of PCI Device Drivers can be created.

12.3.1 PCI Driver Initialization

There are very few differences between a PCI Bus Driver and PCI Device Driver in the entry point of the driver. The file for a driver image must be loaded from some type of media. This could include ROM, FLASH, hard drives, floppy drives, CD-ROM, or even a network connection. Once a driver image has been found, it can be loaded into system memory with the Boot Service **`LoadImage()`**. **`LoadImage()`** loads a PE/COFF formatted image into system memory. A handle is created for the driver, and a Loaded Image Protocol instance is placed on that handle. A handle that contains a Loaded Image Protocol instance is called an *Image Handle*. At this point, the driver has not been started. It is just sitting in memory waiting to be started. Figure 12-7 shows the state of an image handle for a driver after **`LoadImage()`** has been called.

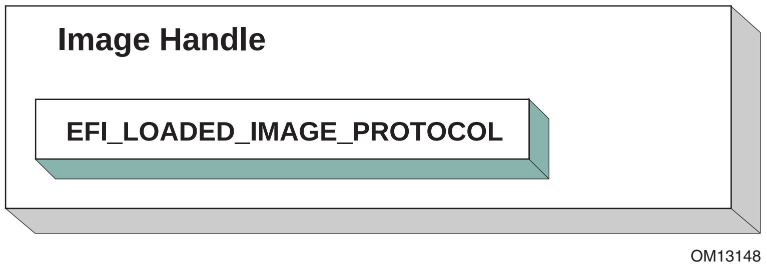


Figure 12-7. Image Handle

After a driver has been loaded with the Boot Service **`LoadImage()`**, it must be started with the Boot Service **`StartImage()`**. This is true of all types of EFI Applications and EFI Drivers that can be loaded and started on an EFI compliant system. The entry point for a driver that follows the EFI Driver Model must follow some strict rules. First, it is not allowed to touch any hardware. Instead, it is only allowed to install protocol instances onto its own *Image Handle*. A driver that follows the EFI Driver Model is *required* to install an instance of the Driver Binding Protocol onto its own *Image Handle*. It may optionally install the Driver Configuration Protocol, the Driver Diagnostics Protocol, or the Component Name Protocol. In addition, if a driver wishes to be unloadable it may optionally update the Loaded Image Protocol to provide its own **`Unload()`** function. Finally, if a driver needs to perform any special operations when the Boot Service **`ExitBootServices()`** is called, it may optionally create an event with a notification function that is triggered when the Boot Service **`ExitBootServices()`** is called. An *Image Handle* that contains a Driver Binding Protocol instance is known as a *Driver Image Handle*. Figure 12-8 shows a possible configuration for the *Image Handle* from Figure 12-7 after the Boot Service **`StartImage()`** has been called.

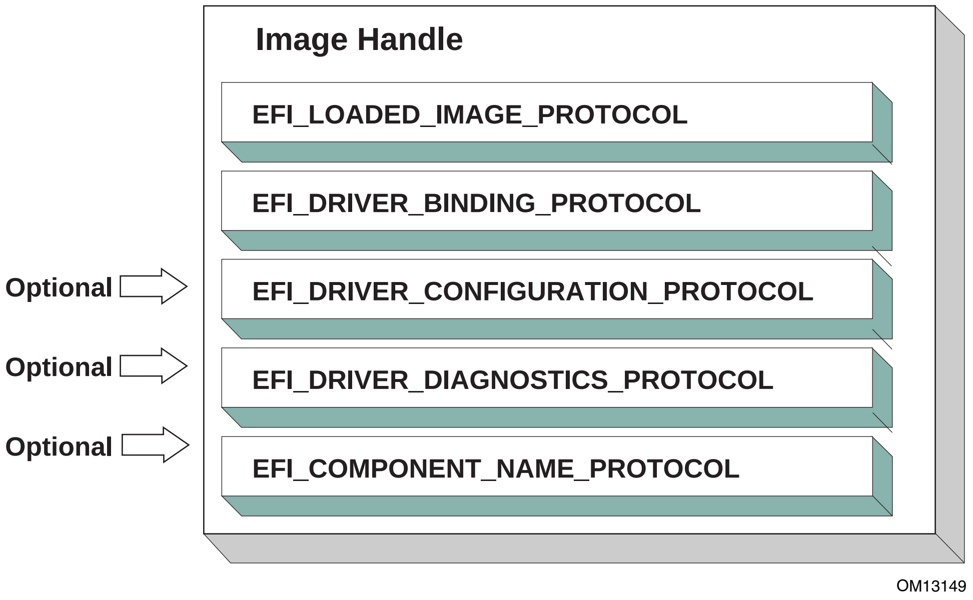


Image Handle
EFI_LOADED_IMAGE_PROTOCOL
EFI_DRIVER_BINDING_PROTOCOL
Optional
EFI_DRIVER_CONFIGURATION_PROTOCOL
Optional
EFI_DRIVER_DIAGNOSTICS_PROTOCOL
Optional
EFI_COMPONENT_NAME_PROTOCOL
OM13149

Figure 12-8. PCI Driver Image Handle

12.3.1.1 Driver Configuration Protocol

If a PCI Bus Driver or a PCI Device Driver requires configuration options, then an **EFI_DRIVER_CONFIGURATION_PROTOCOL** must be installed on the image handle in the entry point for the driver. This protocol contains functions set the configuration information for a controller, validate the current configuration data, and force the configuration data to its default settings. The **EFI_DRIVER_CONFIGURATION_PROTOCOL** must use the standard console devices from the **EFI_SYSTEM_TABLE** to interact with the user. The functions of this protocol will be invoked by a platform management utility. Please see the *EFI Driver Model Specification* for details on the **EFI_DRIVER_CONFIGURATION_PROTOCOL**. Neither this specification, nor the *EFI Driver Model Specification* specifies where configuration data is stored. It is up to the driver writer to decide the appropriate location for configuration data. Some possible locations include a FLASH device or EEPROM device that is attached to the PCI controller, or environment variables accessed through the Runtime Services **GetVariable()** and **SetVariable()**.

12.3.1.2 Driver Diagnostics Protocol

If a PCI Bus Driver or a PCI Device Driver requires diagnostics, then an **EFI_DRIVER_DIAGNOSTICS_PROTOCOL** must be installed on the image handle in the entry point for the driver. This protocol contains functions to perform diagnostics on a controller. The **EFI_DRIVER_DIAGNOSTICS_PROTOCOL** is not allowed to interact with the user. Instead, it must return status information through a buffer. The functions of this protocol will be invoked by a platform management utility. Please see the *EFI Driver Model Specification* for details on the **EFI_DRIVER_DIAGNOSTICS_PROTOCOL**.

12.3.1.3 Component Name Protocol

Both a PCI Bus Driver and a PCI Device Driver are able to produce user readable names for the PCI drivers and/or the set of PCI controllers that the PCI drivers are managing. This is accomplished by installing an instance of the **EFI_COMPONENT_NAME_PROTOCOL** on the image handle of the driver. This protocol can produce driver and controller names in the form of a Unicode string in one of several languages. This protocol can be used by a platform management utility to display user readable names for the drivers and controllers present in a system. Please see the *EFI Driver Model Specification* for details on the **EFI_COMPONENT_NAME_PROTOCOL**.

12.3.2 PCI Bus Drivers

A PCI Bus Driver manages PCI Host Bus Controllers that can contain one or more PCI Root Bridges. Figure 12-9 shows an example of a desktop system that has one PCI Host Bus Controller with one PCI Root Bridge.

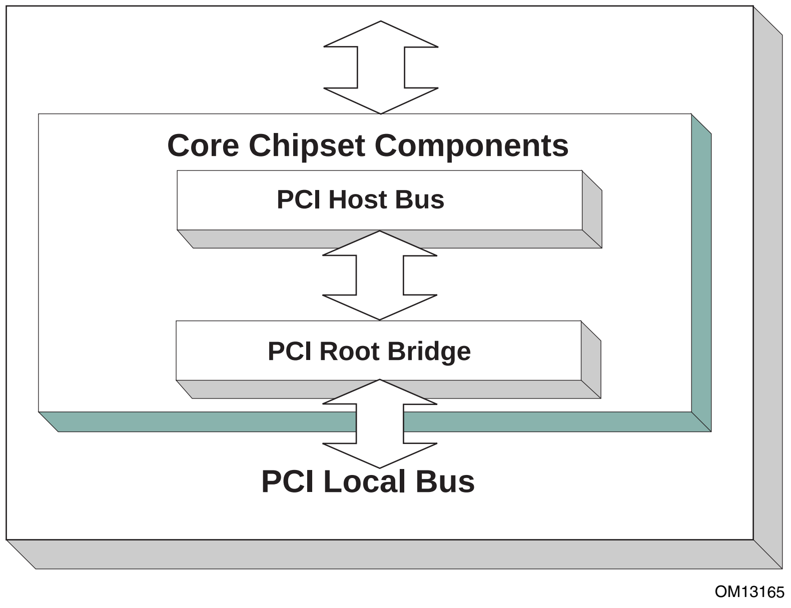


Figure 12-9. PCI Host Bus Controller

The PCI Host Bus Controller in Figure 12-9 is abstracted in software with the PCI Root Bridge I/O Protocol. A PCI Bus Driver will manage handles that contain this protocol. Figure 12-10 shows an example device handle for a PCI Host Bus Controller. It contains a Device Path Protocol instance and a PCI Root Bridge I/O Protocol Instance.

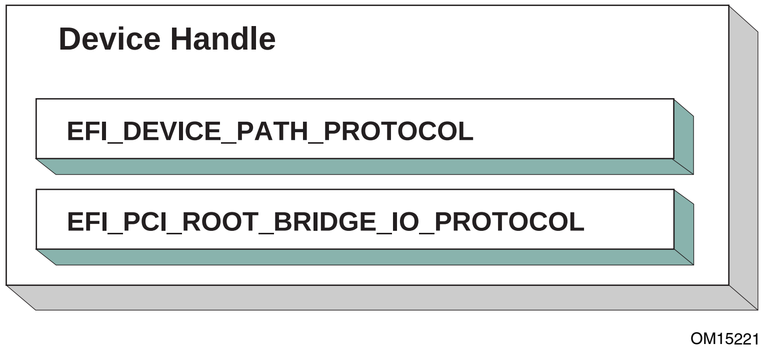


Figure 12-10. Device Handle for a PCI Host Bus Controller

12.3.2.1 Driver Binding Protocol for PCI Bus Drivers

The Driver Binding Protocol contains three services. These are `Supported()`, `Start()`, and `Stop()`. `Supported()` tests to see if the PCI Bus Driver can manage a device handle. A PCI Bus Driver can only manage device handles that contain the Device Path Protocol and the PCI Root Bridge I/O Protocol, so a PCI Bus Driver must look for these two protocols on the device handle that is being tested.

The `Start()` function tells the PCI Bus Driver to start managing a device handle. The device handle should support the protocols shown in Figure 12-10. The PCI Root Bridge I/O Protocols provides access to the PCI I/O, PCI Memory, PCI Prefetchable Memory, and PCI DMA functions. The PCI Controllers behind a PCI Root Bridge may exist on one or more PCI Buses. The standard mechanism for expanding the number of PCI Buses on a single PCI Root Bridge is to use PCI to PCI Bridges. Once a PCI Enumerator configures these bridges, they are invisible to software. As a result, the PCI Bus Driver flattens the PCI Bus hierarchy when it starts managing a device handle that represents a PCI Host Controller. Figure 12-11 shows the physical tree structure for a set of PCI Device denoted by A, B, C, D, and E. Device A and C are PCI to PCI Bridges.

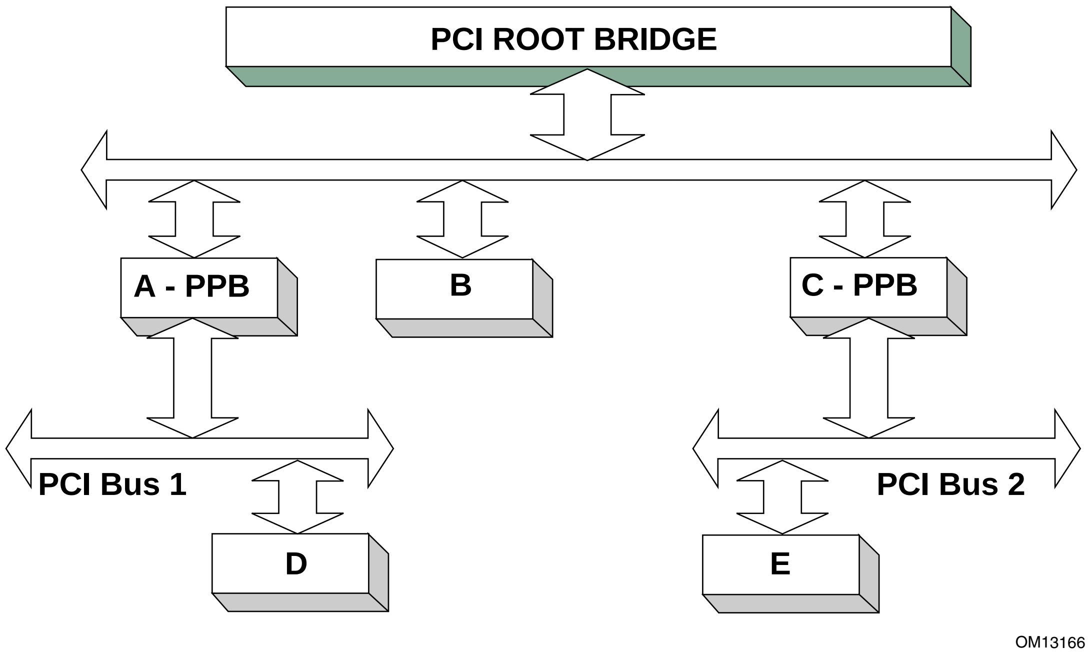


Figure 12-11. Physical PCI Bus Structure

Figure 12-12 shows the tree structure generated by a PCI Bus Driver before and after `Start()` is called. This is a logical view of set of PCI controller, and not a physical view. The physical tree is flattened, so any PCI to PCI bridge devices are invisible. In this example, the PCI Bus Driver finds the five child PCI Controllers on the PCI Bus from Figure 12-11. A device handle is created for every PCI Controller including all the PCI to PCI Bridges. The arrow with the dashed line coming into the PCI Host Bus Controller represents a link to the PCI Host Bus Controller's parent. If the PCI Host Bus Controller is a Root Bus Controller, then it will not have a parent. The PCI Driver Model does not require that a PCI Host Bus Controller be a Root Bus Controller. A PCI Host Bus

Controller can be present at any location in the tree, and the PCI Bus Driver should be able to manage the PCI Host Bus Controller.

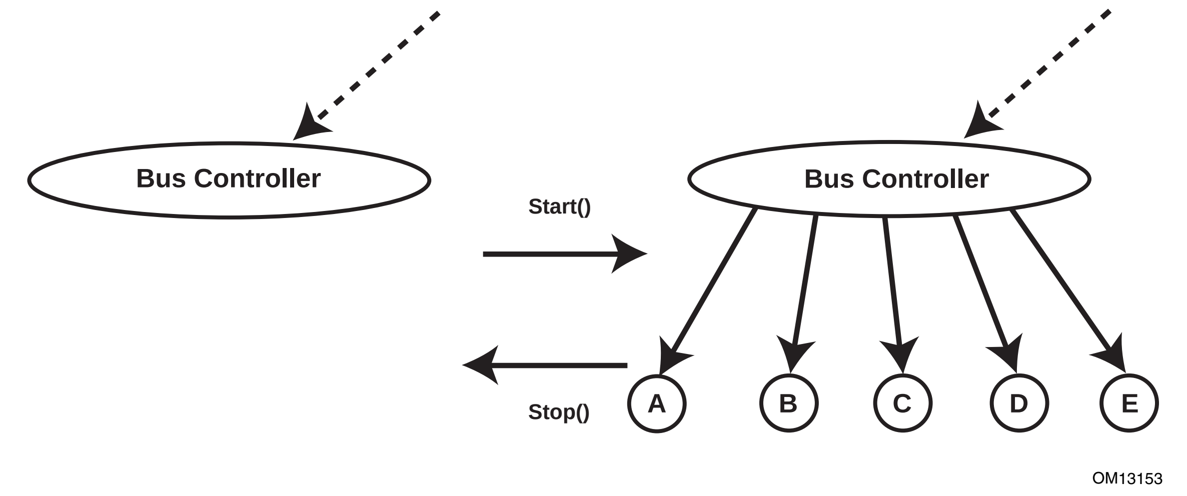


Figure 12-12. Connecting a PCI Bus Driver

The PCI Bus Driver has the option of creating all of its children in one call to `Start()`, or spreading it across several calls to `Start()`. In general, if it is possible to design a bus driver to create one child at a time, it should do so to support the rapid boot capability in the EFI Driver Model. Each of the child device handles created in `Start()` must contain a Device Path Protocol instance, a PCI I/O protocol instance, and optionally a Bus Specific Driver Override Protocol instance. The PCI I/O Protocol is described in Section 12.4. The format of device paths for PCI Controllers is described in Section 2.6, and details on the Bus Specific Driver Override Protocol can be found in the *EFI Driver Model Specification*. Figure 12-13 shows an example child device handle that is created by a PCI Bus Driver for a PCI Controller.

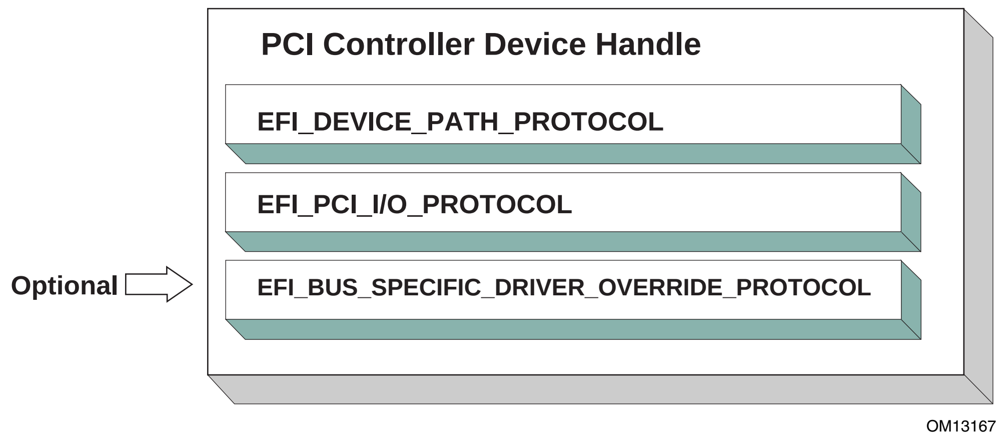

PCI Controller Device Handle
EFI_DEVICE_PATH_PROTOCOL
EFI_PCI_I/O_PROTOCOL
Optional
EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL
OM13167

Figure 12-13. Child Handle Created by a PCI Bus Driver

A PCI Bus Driver must perform several steps to manage a PCI Host Bus Controller, as follows:

- Initialize the PCI Host Bus Controller.
- If the PCI buses have not been initialized by a previous agent, perform PCI Enumeration on all the PCI Root Bridges that the PCI Host Bus Controller contains. This involves assigning a PCI bus number, allocating PCI I/O resources, PCI Memory resources, and PCI Prefetchable Memory resources.
- Discover all the PCI Controllers on all the PCI Root Bridges. If a PCI Controller is a PCI to PCI Bridge, then the I/O, Memory, and Bus Master bits in the Control register of the PCI Configuration Header should be placed in the enabled state. The PCI Bus Driver should not modify the contents of the Control register for any other PCI Controllers. It is a PCI Device Driver's responsibility to enable the I/O, Memory, and Bus Master bits of the Control register as required with a call to the **Attributes()** service when the PCI Device Driver is started. A similar call to the **Attributes()** service should be made when the PCI Device Driver is stopped to disable the I/O, Memory, and Bus Master bits of the Control register.
- Create a device handle for each PCI Controller found. If a request is being made to start only one PCI Controller, then only create one device handle.
- Install a Device Path Protocol instance and a PCI I/O Protocol instance on the device handle created for each PCI Controller.
- If the PCI Controller has a PCI Option ROM, then allocate a memory buffer that is the same size as the PCI Option ROM, and copy the PCI Option ROM contents to the memory buffer.
- If the PCI Option ROM contains any EFI Drivers, then attach a Bus Specific Driver Override Protocol to the device handle of the PCI Controller that is associated with the PCI Option ROM.

The **Stop()** function tells the PCI Bus Driver to stop managing a PCI Host Bus Controller. The **Stop()** function can destroy one or more of the device handles that were created on a previous call to **Start()**. If all of the child device handles have been destroyed, then **Stop()** will place the PCI Host Bus Controller in a quiescent state. The functionality of **Stop()** mirrors **Start()**, as follows:

1. Complete all outstanding transactions to the PCI Host Bus Controller.
2. If the PCI Host Bus Controller is being stopped, then place it in a quiescent state.
3. If one or more child handles are being destroyed, then:
   a. Uninstall all the protocols from the device handles for the PCI Controllers found in **Start()**.
   b. Free any memory buffers allocated for PCI Option ROMs.
   c. Destroy the device handles for the PCI controllers created in **Start()**.

12.3.2.2 PCI Enumeration

The PCI Enumeration process is a platform-specific operation that depends on the properties of the chipset that produces the PCI bus. As a result, details on PCI Enumeration are outside the scope of this document. A PCI Bus Driver requires that PCI Enumeration has been performed, so it either needs to have been done prior to the PCI Bus Driver starting, or it must be part of the PCI Bus Driver's implementation.

12.3.3 PCI Device Drivers

PCI Device Drivers manage PCI Controllers. Device handles for PCI Controllers are created by PCI Bus Drivers. A PCI Device Driver is not allowed to create any new device handles. Instead, it attaches protocol instance to the device handle of the PCI Controller. These protocol instances are I/O abstractions that allow the PCI Controller to be used in the preboot environment. The most common I/O abstractions are used to boot an EFI compliant OS.

12.3.3.1 Driver Binding Protocol for PCI Device Drivers

The Driver Binding Protocol contains three services. These are **`Supported()`**, **`Start()`**, and **`Stop()`**. **`Supported()`** tests to see if the PCI Device Driver can manage a device handle. A PCI Device Driver can only manage device handles that contain the Device Path Protocol and the PCI I//O Protocol, so a PCI Device Driver must look for these two protocols on the device handle that is being tested. In addition, it needs to check to see if the device handle represents a PCI Controller that the PCI Device Driver knows how to manage. This is typically done by using the services of the PCI I/O Protocol to read the PCI Configuration Header for the PCI Controller, and looking at the *`VendorId`*, *`DeviceId`*, and *`SubsystemId`* fields.

The **`Start()`** function tells the PCI Device Driver to start managing a PCI Controller. A PCI Device Driver is not allowed to create any new device handles. Instead, it installs one or more addition protocol instances on the device handle for the PCI Controller. A PCI Device Driver is not allowed to modify the resources allocated to a PCI Controller. These resource allocations are owned by the PCI Bus Driver or some other firmware component that initialized the PCI Bus prior to the execution of the PCI Bus Driver. This means that the PCI BARs (Base Address Registers) and the configuration of any PCI to PCI bridge controllers must not be modified by a PCI Device Driver. A PCI Bus Driver will leave a PCI Device in a disabled state. It is a PCI Device Driver's responsibility to call **`Attributes()`** to enable the I/O, Memory, and Bus Master decodes.

The `Stop()` function mirrors the `Start()` function, so the `Stop()` function completes any outstanding transactions to the PCI Controller and removes the protocol interfaces that were installed in `Start()`. Figure 12-14 shows the device handle for a PCI Controller before and after `Start()` is called. In this example, a PCI Device Driver is adding the Block I/O Protocol to the device handle for the PCI Controller. It is also a PCI Device Driver's responsibility to disable the I/O, Memory, and Bus Master decodes by calling `Attributes()`.

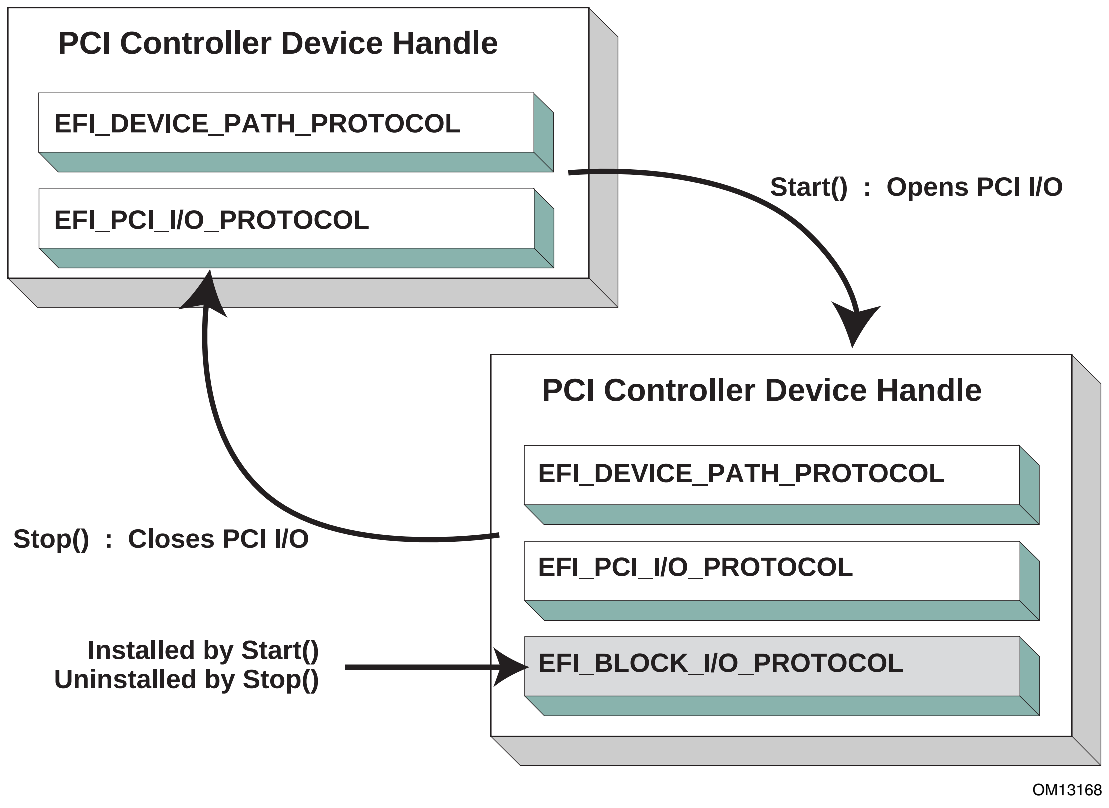


Figure 12-14. Connecting a PCI Device Driver

12.4 EFI PCI I/O Protocol

This section provides a detailed description of the **EFI_PCI_IO_PROTOCOL**. This protocol is used by code, typically drivers, running in the EFI boot services environment to access memory and I/O on a PCI controller. In particular, functions for managing devices on PCI buses are defined here.

The interfaces provided in the **EFI_PCI_IO_PROTOCOL** are for performing basic operations to memory, I/O, and PCI configuration space. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources. The main goal of this protocol is to provide an abstraction that simplifies the writing of device drivers for PCI devices. This goal is accomplished by providing the following features:

- A driver model that does not require the driver to search the PCI busses for devices to manage. Instead, drivers are provided the location of the device to manage or have the capability to be notified when a PCI controller is discovered.
- A device driver model that abstracts the I/O addresses, Memory addresses, and PCI Configuration addresses from the PCI device driver. Instead, BAR (Base Address Register) relative addressing is used for I/O and Memory accesses, and device relative addressing is used for PCI Configuration accesses. The BAR relative addressing is specified in the PCI I/O services as a BAR index. A PCI controller may contain a combination of 32-bit and 64-bit BARs. The BAR index represents the logical BAR number in the standard PCI configuration header starting from the first BAR. The BAR index does not represent an offset into the standard PCI Configuration Header because those offsets will vary depending on the combination and order of 32-bit and 64-bit BARs.
- The Device Path for the PCI device can be obtained from the same device handle that the **EFI_PCI_IO_PROTOCOL** resides.
- The PCI Segment, PCI Bus Number, PCI Device Number, and PCI Function Number of the PCI device if they are required. The general idea is to abstract these details away from the PCI device driver. However, if these details are required, then they are available.
- Details on any nonstandard address decoding that is not covered by the PCI device's Base Address Registers.
- Access to the PCI Root Bridge I/O Protocol for the PCI Host Bus for which the PCI device is a member.
- A copy of the PCI Option ROM if it is present in system memory.
- Functions to perform bus mastering DMA. This includes both packet based DMA and common buffer DMA.

EFI_PCI_IO_PROTOCOL

Summary

Provides the basic Memory, I/O, PCI configuration, and DMA interfaces that a driver uses to access its PCI controller.

GUID

```
#define EFI_PCI_IO_PROTOCOL_GUID  \
{0x4cf5b200,0x68b8,0x4ca5,0x9e,0xec,0xb2,0x3e,0x3f,0x50,0x2,0x9a}
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_IO_PROTOCOL {
  EFI_PCI_IO_PROTOCOL_POLL_IO_MEM          PollMem;
  EFI_PCI_IO_PROTOCOL_POLL_IO_MEM          PollIo;
  EFI_PCI_IO_PROTOCOL_ACCESS               Mem;
  EFI_PCI_IO_PROTOCOL_ACCESS               Io;
  EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS        Pci;
  EFI_PCI_IO_PROTOCOL_COPY_MEM             CopyMem;
  EFI_PCI_IO_PROTOCOL_MAP                  Map;
  EFI_PCI_IO_PROTOCOL_UNMAP                Unmap;
  EFI_PCI_IO_PROTOCOL_ALLOCATE_BUFFER      AllocateBuffer;
  EFI_PCI_IO_PROTOCOL_FREE_BUFFER          FreeBuffer;
  EFI_PCI_IO_PROTOCOL_FLUSH                Flush;
  EFI_PCI_IO_PROTOCOL_GET_LOCATION         GetLocation;
  EFI_PCI_IO_PROTOCOL_ATTRIBUTES           Attributes;
  EFI_PCI_IO_PROTOCOL_GET_BAR_ATTRIBUTES   GetBarAttributes;
  EFI_PCI_IO_PROTOCOL_SET_BAR_ATTRIBUTES   SetBarAttributes;
  UINT64                                   RomSize;
  VOID                                     *RomImage;
} EFI_PCI_IO_PROTOCOL;
```

Parameters

PollMem	Polls an address in PCI memory space until an exit condition is met, or a timeout occurs. See the **PollMem()** function description.
PollIo	Polls an address in PCI I/O space until an exit condition is met, or a timeout occurs. See the **PollIo()** function description.
Mem.Read	Allows BAR relative reads to PCI memory space. See the **Mem.Read()** function description.
Mem.Write	Allows BAR relative writes to PCI memory space. See the **Mem.Write()** function description.

Io.Read	Allows BAR relative reads to PCI I/O space. See the **Io.Read()** function description.
Io.Write	Allows BAR relative writes to PCI I/O space. See the **Io.Write()** function description.
Pci.Read	Allows PCI controller relative reads to PCI configuration space. See the **Pci.Read()** function description.
Pci.Write	Allows PCI controller relative writes to PCI configuration space. See the **Pci.Write()** function description.
CopyMem	Allows one region of PCI memory space to be copied to another region of PCI memory space. See the **CopyMem()** function description.
Map	Provides the PCI controller–specific address needed to access system memory for DMA. See the **Map()** function description.
Unmap	Releases any resources allocated by **Map()**. See the **Unmap()** function description.
AllocateBuffer	Allocates pages that are suitable for a common buffer mapping. See the **AllocateBuffer()** function description.
FreeBuffer	Frees pages that were allocated with **AllocateBuffer()**. See the **FreeBuffer()** function description.
Flush	Flushes all PCI posted write transactions to system memory. See the **Flush()** function description.
GetLocation	Retrieves this PCI controller's current PCI bus number, device number, and function number. See the **GetLocation()** function description.
Attributes	Performs an operation on the attributes that this PCI controller supports. The operations include getting the set of supported attributes, retrieving the current attributes, setting the current attributes, enabling attributes, and disabling attributes. See the **Attributes()** function description.
GetBarAttributes	Gets the attributes that this PCI controller supports setting on a BAR using **SetBarAttributes()**, and retrieves the list of resource descriptors for a BAR. See the **GetBarAttributes()** function description.
SetBarAttributes	Sets the attributes for a range of a BAR on a PCI controller. See the **SetBarAttributes()** function description.
RomSize	The size, in bytes, of the ROM image.

RomImage	A pointer to the in memory copy of the ROM image. The PCI Bus Driver is responsible for allocating memory for the ROM image, and copying the contents of the ROM to memory. The contents of this buffer are either from the PCI option ROM that can be accessed through the ROM BAR of the PCI controller, or it is from a platform-specific location. The **Attributes()** function can be used to determine from which of these two sources the *RomImage* buffer was initialized.

Related Definitions

```
//*******************************************************
// EFI_PCI_IO_PROTOCOL_WIDTH
//*******************************************************
typedef enum {
  EfiPciIoWidthUint8,
  EfiPciIoWidthUint16,
  EfiPciIoWidthUint32,
  EfiPciIoWidthUint64,
  EfiPciIoWidthFifoUint8,
  EfiPciIoWidthFifoUint16,
  EfiPciIoWidthFifoUint32,
  EfiPciIoWidthFifoUint64,
  EfiPciIoWidthFillUint8,
  EfiPciIoWidthFillUint16,
  EfiPciIoWidthFillUint32,
  EfiPciIoWidthFillUint64,
  EfiPciIoWidthMaximum
} EFI_PCI_IO_PROTOCOL_WIDTH;

#define EFI_PCI_IO_PASS_THROUGH_BAR    0xff

//*******************************************************
// EFI_PCI_IO_PROTOCOL_POLL_IO_MEM
//*******************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_POLL_IO_MEM) (
  IN  struct EFI_PCI_IO_PROTOCOL  *This,
  IN  EFI_PCI_IO_PROTOCOL_WIDTH   Width,
  IN  UINT8                       BarIndex,
  IN  UINT64                      Offset,
  IN  UINT64                      Mask,
  IN  UINT64                      Value,
  IN  UINT64                      Delay,
  OUT UINT64                      *Result
  );
```

```
//*******************************************************
// EFI_PCI_IO_PROTOCOL_IO_MEM
//*******************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_IO_MEM) (
  IN      EFI_PCI_IO_PROTOCOL         *This,
  IN      EFI_PCI_IO_PROTOCOL_WIDTH   Width,
  IN      UINT8                       BarIndex,
  IN      UINT64                      Offset,
  IN      UINTN                       Count,
  IN OUT VOID                         *Buffer
  );

//*******************************************************
// EFI_PCI_IO_PROTOCOL_ACCESS
//*******************************************************
typedef struct {
  EFI_PCI_IO_PROTOCOL_IO_MEM  Read;
  EFI_PCI_IO_PROTOCOL_IO_MEM  Write;
} EFI_PCI_IO_PROTOCOL_ACCESS;

//*******************************************************
// EFI_PCI_IO_PROTOCOL_CONFIG
//*******************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_CONFIG) (
  IN      EFI_PCI_IO_PROTOCOL         *This,
  IN      EFI_PCI_IO_PROTOCOL_WIDTH   Width,
  IN      UINT32                      Offset,
  IN      UINTN                       Count,
  IN OUT VOID                         *Buffer
  );

//*******************************************************
// EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS
//*******************************************************
typedef struct {
  EFI_PCI_IO_PROTOCOL_CONFIG  Read;
  EFI_PCI_IO_PROTOCOL_CONFIG  Write;
} EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS;
```

```
//*******************************************************
// EFI PCI I/O Protocol Attribute bits, EFI PCI I/O Protocol
//*******************************************************
#define EFI_PCI_IO_ATTRIBUTE_ISA_MOTHERBOARD_IO    0x0001
#define EFI_PCI_IO_ATTRIBUTE_ISA_IO                0x0002
#define EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO        0x0004
#define EFI_PCI_IO_ATTRIBUTE_VGA_MEMORY            0x0008
#define EFI_PCI_IO_ATTRIBUTE_VGA_IO                0x0010
#define EFI_PCI_IO_ATTRIBUTE_IDE_PRIMARY_IO        0x0020
#define EFI_PCI_IO_ATTRIBUTE_IDE_SECONDARY_IO      0x0040
#define EFI_PCI_IO_ATTRIBUTE_MEMORY_WRITE_COMBINE  0x0080
#define EFI_PCI_IO_ATTRIBUTE_IO                    0x0100
#define EFI_PCI_IO_ATTRIBUTE_MEMORY                0x0200
#define EFI_PCI_IO_ATTRIBUTE_BUS_MASTER            0x0400
#define EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED         0x0800
#define EFI_PCI_IO_ATTRIBUTE_MEMORY_DISABLE        0x1000
#define EFI_PCI_IO_ATTRIBUTE_EMBEDDED_DEVICE       0x2000
#define EFI_PCI_IO_ATTRIBUTE_EMBEDDED_ROM          0x4000
#define EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE    0x8000
```

EFI_PCI_IO_ATTRIBUTE_ISA_MOTHERBOARD_IO

If this bit is set, then the PCI I/O cycles between 0x00000000 and 0x000000FF are forwarded to the PCI controller. This bit is used to forward I/O cycles for ISA motherboard devices. If this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller are configured to forward these PCI I/O cycles.

EFI_PCI_IO_ATTRIBUTE_ISA_IO

If this bit is set, then the PCI I/O cycles between 0x100 and 0x3FF are forwarded to the PCI controller using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O cycles for legacy ISA devices. If this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller are configured to forward these PCI I/O cycles.

EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO

If this bit is set, then the PCI I/O write cycles for 0x3C6, 0x3C8, and 0x3C9 are forwarded to the PCI controller using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O write cycles to the VGA palette registers on a PCI controller. If this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller are configured to forward these PCI I/O cycles.

EFI_PCI_IO_ATTRIBUTE_VGA_MEMORY

If this bit is set, then the PCI memory cycles between 0xA0000 and 0xBFFFF are forwarded to the PCI controller. This bit is used to forward memory cycles for a VGA frame buffer on a PCI controller. If this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller are configured to forward these PCI Memory cycles.

EFI_PCI_IO_ATTRIBUTE_VGA_IO

If this bit is set, then the PCI I/O cycles in the ranges 0x3B0-0x3BB and 0x3C0-0x3DF are forwarded to the PCI controller using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and the address bits 16..31 must be zero. This bit is used to forward I/O cycles for a VGA controller to a PCI controller. If this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller are configured to forward these PCI I/O cycles. Since **EFI_PCI_IO_ATTRIBUTE_VGA_IO** also includes the I/O range described by **EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_ IO**, the **EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO** bit is ignored if **EFI_PCI_IO_ATTRIBUTE_VGA_IO** is set.

EFI_PCI_IO_ATTRIBUTE_IDE_PRIMARY_IO

If this bit is set, then the PCI I/O cycles in the ranges 0x1F0-0x1F7 and 0x3F6-0x3F7 are forwarded to a PCI controller using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O cycles for a Primary IDE controller to a PCI controller. If this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller are configured to forward these PCI I/O cycles.

EFI_PCI_IO_ATTRIBUTE_IDE_SECONDARY_IO

If this bit is set, then the PCI I/O cycles in the ranges 0x170-0x177 and 0x376-0x377 are forwarded to a PCI controller using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O cycles for a Secondary IDE controller to a PCI controller. If this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller are configured to forward these PCI I/O cycles.

EFI_PCI_IO_ATTRIBUTE_MEMORY_WRITE_COMBINE

If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is accessed in a write combining mode. This bit is used to improve the write performance to a memory buffer on a PCI controller. By default, PCI memory ranges are not accessed in a write combining mode.

EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED

If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is accessed in a cached mode. By default, PCI memory ranges are accessed noncached.

EFI_PCI_IO_ATTRIBUTE_IO

If this bit is set, then the PCI device will decode the PCI I/O cycles that the device is configured to decode.

EFI_PCI_IO_ATTRIBUTE_MEMORY

If this bit is set, then the PCI device will decode the PCI Memory cycles that the device is configured to decode.

EFI_PCI_IO_ATTRIBUTE_BUS_MASTER

If this bit is set, then the PCI device is allowed to act as a bus master on the PCI bus.

EFI_PCI_IO_ATTRIBUTE_MEMORY_DISABLE

If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is disabled, and can no longer be accessed. By default, all PCI memory ranges are enabled.

EFI_PCI_IO_ATTRIBUTE_EMBEDDED_DEVICE

If this bit is set, then the PCI controller is an embedded device that is typically a component on the system board. If this bit is clear, then this PCI controller is part of an adapter that is populating one of the systems PCI slots.

EFI_PCI_IO_ATTRIBUTE_EMBEDDED_ROM

If this bit is set, then the PCI option ROM described by the *RomImage* and *RomSize* fields is not from ROM BAR of the PCI controller. If this bit is clear, then the *RomImage* and *RomSize* fields were initialized based on the PCI option ROM found through the ROM BAR of the PCI controller.

EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE

If this bit is set, then the PCI controller is capable of producing PCI Dual Address Cycles, so it is able to access a 64-bit address space. If this bit is not set, then the PCI controller is not capable of producing PCI Dual Address Cycles, so it is only able to access a 32-bit address space.

```
//*******************************************************
// EFI_PCI_IO_PROTOCOL_OPERATION
//*******************************************************
typedef enum {
  EfiPciIoOperationBusMasterRead,
  EfiPciIoOperationBusMasterWrite,
  EfiPciIoOperationBusMasterCommonBuffer,
  EfiPciIoOperationMaximum
} EFI_PCI_IO_PROTOCOL_OPERATION;
```

EfiPciIoOperationBusMasterRead	A read operation from system memory by a bus master.
EfiPciIoOperationBusMasterWrite	A write operation to system memory by a bus master.
EfiPciIoOperationBusMasterCommonBuffer	Provides both read and write access to system memory by both the processor and a bus master. The buffer is coherent from both the processor's and the bus master's point of view.

Description

The **EFI_PCI_IO_PROTOCOL** provides the basic Memory, I/O, PCI configuration, and DMA interfaces that are used to abstract accesses to PCI controllers. There is one **EFI_PCI_IO_PROTOCOL** instance for each PCI controller on a PCI bus. A device driver that wishes to manage a PCI controller in a system will have to retrieve the **EFI_PCI_IO_PROTOCOL** instance that is associated with the PCI controller. A device handle for a PCI controller will minimally contain an **EFI_DEVICE_PATH_PROTOCOL** instance and an **EFI_PCI_IO_PROTOCOL** instance.

Bus mastering PCI controllers can use the DMA services for DMA operations. There are three basic types of bus mastering DMA that is supported by this protocol. These are DMA reads by a bus master, DMA writes by a bus master, and common buffer DMA. The DMA read and write operations may need to be broken into smaller chunks. The caller of **Map()** must pay attention to the number of bytes that were mapped, and if required, loop until the entire buffer has been transferred. The following is a list of the different bus mastering DMA operations that are supported, and the sequence of **EFI_PCI_IO_PROTOCOL** interfaces that are used for each DMA operation type.

DMA Bus Master Read Operation

- Call `Map()` for `EfiPciIoOperationBusMasterRead`.
- Program the DMA Bus Master with the *`DeviceAddress`* returned by `Map()`.
- Start the DMA Bus Master.
- Wait for DMA Bus Master to complete the read operation.
- Call `Unmap()`.

DMA Bus Master Write Operation

- Call `Map()` for `EfiPciOperationBusMasterWrite`.
- Program the DMA Bus Master with the *`DeviceAddress`* returned by `Map()`.
- Start the DMA Bus Master.
- Wait for DMA Bus Master to complete the write operation.
- Perform a PCI controller specific read transaction to flush all PCI write buffers (See *PCI Specification* Section 3.2.5.2) .
- Call `Flush()`.
- Call `Unmap()`.

DMA Bus Master Common Buffer Operation

- Call `AllocateBuffer()` to allocate a common buffer.
- Call `Map()` for `EfiPciIoOperationBusMasterCommonBuffer`.
- Program the DMA Bus Master with the *`DeviceAddress`* returned by `Map()`.
- The common buffer can now be accessed equally by the processor and the DMA bus master.
- Call `Unmap()`.
- Call `FreeBuffer()`.

EFI_PCI_IO_PROTOCOL.PollMem()

Summary

Reads from the memory space of a PCI controller. Returns when either the polling exit criteria is satisfied or after a defined duration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_POLL_IO_MEM) (
  IN  struct EFI_PCI_IO_PROTOCOL   *This,
  IN  EFI_PCI_IO_PROTOCOL_WIDTH    Width,
  IN  UINT8                        BarIndex,
  IN  UINT64                       Offset,
  IN  UINT64                       Mask,
  IN  UINT64                       Value,
  IN  UINT64                       Delay,
  OUT UINT64                       *Result
);
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Width	Signifies the width of the memory operations. Type **EFI_PCI_IO_PROTOCOL_WIDTH** is defined in Section 12.4.
BarIndex	The BAR index of the standard PCI Configuration header to use as the base address for the memory operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value **EFI_PCI_IO_PASS_THROUGH_BAR** can be used to bypass the BAR relative addressing and pass *Offset* to the PCI Root Bridge I/O Protocol unchanged. Type **EFI_PCI_IO_PASS_THROUGH_BAR** is defined in Section 12.4.
Offset	The offset within the selected BAR to start the memory operation.
Mask	Mask used for the polling criteria. Bytes above *Width* in *Mask* are ignored. The bits in the bytes below *Width* which are zero in *Mask* are ignored when polling the memory address.

Value	The comparison value used for the polling exit criteria.
Delay	The number of 100 ns units to poll. Note that timer available may be of poorer granularity.
Result	Pointer to the last value read from the memory location.

Description

This function provides a standard way to poll a PCI memory location. A PCI memory read operation is performed at the PCI memory address specified by *BarIndex* and *Offset* for the width specified by *Width*. The result of this PCI memory read operation is stored in *Result*. This PCI memory read operation is repeated until either a timeout of *Delay* 100 ns units has expired, or (*Result & Mask)* is equal to *Value*.

This function will always perform at least one memory access no matter how small *Delay* may be. If *Delay* is 0, then *Result* will be returned with a status of **EFI_SUCCESS** even if *Result* does not match the exit criteria. If *Delay* expires, then **EFI_TIMEOUT** is returned.

If *Width* is not **EfiPciIoWidthUint8**, **EfiPciIoWidthUint16**, **EfiPciIoWidthUint32**, or **EfiPciIoWidthUint64**, then **EFI_INVALID_PARAMETER** is returned.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI controller on a platform might require. For example on some platforms, width requests of **EfiPciIoWidthUint64** do not work.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. However, if the memory mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Status Codes Returned

EFI_SUCCESS	The last data returned from the access matched the poll exit criteria.
EFI_INVALID_PARAMETER	*Width* is invalid.
EFI_INVALID_PARAMETER	*Result* is **NULL**.
EFI_UNSUPPORTED	*BarIndex* not valid for this PCI controller.
EFI_UNSUPPORTED	*Offset* is not valid for the *BarIndex* of this PCI controller.
EFI_TIMEOUT	*Delay* expired before a match occurred.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_IO_PROTOCOL.PollIo()

Summary

Reads from the I/O space of a PCI controller. Returns when either the polling exit criteria is satisfied or after a defined duration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_POLL_IO_MEM) (
  IN  struct EFI_PCI_IO_PROTOCOL   *This,
  IN  EFI_PCI_IO_PROTOCOL_WIDTH    Width,
  IN  UINT8                        BarIndex,
  IN  UINT64                       Offset,
  IN  UINT64                       Mask,
  IN  UINT64                       Value,
  IN  UINT64                       Delay,
  OUT UINT64                       *Result
);
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Width	Signifies the width of the I/O operations. Type **EFI_PCI_IO_PROTOCOL_WIDTH** is defined in Section 12.4.
BarIndex	The BAR index of the standard PCI Configuration header to use as the base address for the I/O operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value **EFI_PCI_IO_PASS_THROUGH_BAR** can be used to bypass the BAR relative addressing and pass *Offset* to the PCI Root Bridge I/O Protocol unchanged. Type **EFI_PCI_IO_PASS_THROUGH_BAR** is defined in Section 12.4.
Offset	The offset within the selected BAR to start the I/O operation.
Mask	Mask used for the polling criteria. Bytes above *Width* in *Mask* are ignored. The bits in the bytes below *Width* which are zero in *Mask* are ignored when polling the I/O address.

Value	The comparison value used for the polling exit criteria.
Delay	The number of 100 ns units to poll. Note that timer available may be of poorer granularity.
Result	Pointer to the last value read from the memory location.

Description

This function provides a standard way to poll a PCI I/O location. A PCI I/O read operation is performed at the PCI I/O address specified by *BarIndex* and *Offset* for the width specified by *Width*. The result of this PCI I/O read operation is stored in *Result*. This PCI I/O read operation is repeated until either a timeout of *Delay* 100 ns units has expired, or (*Result & Mask)* is equal to *Value*.

This function will always perform at least one I/O access no matter how small *Delay* may be. If *Delay* is 0, then *Result* will be returned with a status of **EFI_SUCCESS** even if *Result* does not match the exit criteria. If *Delay* expires, then **EFI_TIMEOUT** is returned.

If *Width* is not **EfiPciIoWidthUint8**, **EfiPciIoWidthUint16**, **EfiPciIoWidthUint32**, or **EfiPciIoWidthUint64**, then **EFI_INVALID_PARAMETER** is returned.

The I/O operations are carried out exactly as requested. The caller is responsible satisfying any alignment and I/O width restrictions that the PCI controller on a platform might require. For example on some platforms, width requests of **EfiPciIoWidthUint64** do not work.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Status Codes Returned

EFI_SUCCESS	The last data returned from the access matched the poll exit criteria.
EFI_INVALID_PARAMETER	*Width* is invalid.
EFI_INVALID_PARAMETER	*Result* is **NULL**.
EFI_UNSUPPORTED	*BarIndex* not valid for this PCI controller.
EFI_UNSUPPORTED	*Offset* is not valid for the PCI BAR specified by *BarIndex*.
EFI_TIMEOUT	*Delay* expired before a match occurred.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_IO_PROTOCOL.Mem.Read()
EFI_PCI_IO_PROTOCOL.Mem.Write()

Summary

Enable a PCI driver to access PCI controller registers in the PCI memory space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_MEM) (
  IN     EFI_PCI_IO_PROTOCOL        *This,
  IN     EFI_PCI_IO_PROTOCOL_WIDTH  Width,
  IN     UINT8                      BarIndex,
  IN     UINT64                     Offset,
  IN     UINTN                      Count,
  IN OUT VOID                       *Buffer
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Width	Signifies the width of the memory operations. Type **EFI_PCI_IO_PROTOCOL_WIDTH** is defined in Section 12.4.
BarIndex	The BAR index of the standard PCI Configuration header to use as the base address for the memory operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value **EFI_PCI_IO_PASS_THROUGH_BAR** can be used to bypass the BAR relative addressing and pass *Offset* to the PCI Root Bridge I/O Protocol unchanged. Type **EFI_PCI_IO_PASS_THROUGH_BAR** is defined in Section 12.4.
Offset	The offset within the selected BAR to start the memory operation.
Count	The number of memory operations to perform. Bytes moved is *Width* size * *Count*, starting at *Offset*.
Buffer	For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description

The **Mem.Read()**, and **Mem.Write()** functions enable a driver to access controller registers in the PCI memory space.

The I/O operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues which the bus, device, platform, or type of I/O might require. For example on some platforms, width requests of **EfiPciIoWidthUint64** do not work.

If *Width* is **EfiPciIoWidthUint8**, **EfiPciIoWidthUint16**, **EfiPciIoWidthUint32**, or **EfiPciIoWidthUint64**, then both *Address* and *Buffer* are incremented for each of the *Count* operations performed.

If *Width* is **EfiPciIoWidthFifoUint8**, **EfiPciIoWidthFifoUint16**, **EfiPciIoWidthFifoUint32**, or **EfiPciIoWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiPciIoWidthFillUint8**, **EfiPciIoWidthFillUint16**, **EfiPciIoWidthFillUint32**, or **EfiPciIoWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the *PCI Specification*. However, if the memory-mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PCI controller.
EFI_INVALID_PARAMETER	*Width* is invalid.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_UNSUPPORTED	*BarIndex* not valid for this PCI controller.
EFI_UNSUPPORTED	The address range specified by *Offset*, *Width*, and *Count* is not valid for the PCI BAR specified by *BarIndex*.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_IO_PROTOCOL.Io.Read()
EFI_PCI_IO_PROTOCOL.Io.Write()

Summary

Enable a PCI driver to access PCI controller registers in the PCI I/O space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_MEM) (
  IN     EFI_PCI_IO_PROTOCOL         *This,
  IN     EFI_PCI_IO_PROTOCOL_WIDTH   Width,
  IN     UINT8                       BarIndex,
  IN     UINT64                      Offset,
  IN     UINTN                       Count,
  IN OUT VOID                        *Buffer
);
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Width	Signifies the width of the memory operations. Type **EFI_PCI_IO_PROTOCOL_WIDTH** is defined in Section 12.4.
BarIndex	The BAR index in the standard PCI Configuration header to use as the base address for the I/O operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value **EFI_PCI_IO_PASS_THROUGH_BAR** can be used to bypass the BAR relative addressing and pass *Offset* to the PCI Root Bridge I/O Protocol unchanged. Type **EFI_PCI_IO_PASS_THROUGH_BAR** is defined in Section 12.4.
Offset	The offset within the selected BAR to start the I/O operation.
Count	The number of I/O operations to perform. Bytes moved is *Width* size * *Count*, starting at *Offset*.
Buffer	For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description

The **Io.Read()**, and **Io.Write()** functions enable a driver to access PCI controller registers in PCI I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues which the bus, device, platform, or type of I/O might require. For example on some platforms, width requests of **EfiPciIoWidthUint64** do not work.

If *Width* is **EfiPciIoWidthUint8**, **EfiPciIoWidthUint16**, **EfiPciIoWidthUint32**, or **EfiPciIoWidthUint64**, then both *Address* and *Buffer* are incremented for each of the *Count* operations performed.

If *Width* is **EfiPciIoWidthFifoUint8**, **EfiPciIoWidthFifoUint16**, **EfiPciIoWidthFifoUint32**, or **EfiPciIoWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiPciIoWidthFillUint8**, **EfiPciIoWidthFillUint16**, **EfiPciIoWidthFillUint32**, or **EfiPciIoWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PCI controller.
EFI_INVALID_PARAMETER	*Width* is invalid.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_UNSUPPORTED	*BarIndex* not valid for this PCI controller.
EFI_UNSUPPORTED	The address range specified by *Offset*, *Width*, and *Count* is not valid for the PCI BAR specified by *BarIndex*.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_IO_PROTOCOL.Pci.Read()
EFI_PCI_IO_PROTOCOL.Pci.Write()

Summary

Enable a PCI driver to access PCI controller registers in PCI configuration space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_CONFIG) (
  IN      EFI_PCI_IO_PROTOCOL         *This,
  IN      EFI_PCI_IO_PROTOCOL_WIDTH   Width,
  IN      UINT32                      Offset,
  IN      UINTN                       Count,
  IN OUT  VOID                        *Buffer
);
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Width	Signifies the width of the memory operations. Type **EFI_PCI_IO_PROTOCOL_WIDTH** is defined in Section 12.4.
Offset	The offset within the PCI configuration space for the PCI controller.
Count	The number of PCI configuration operations to perform. Bytes moved is *Width* size * *Count*, starting at *Offset*.
Buffer	For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description

The **`Pci.Read()`** and **`Pci.Write()`** functions enable a driver to access PCI configuration registers for the PCI controller.

The PCI Configuration operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues which the bus, device, platform, or type of I/O might require. For example on some platforms, width requests of **`EfiPciIoWidthUint64`** do not work.

If *`Width`* is **`EfiPciIoWidthUint8`**, **`EfiPciIoWidthUint16`**, **`EfiPciIoWidthUint32`**, or **`EfiPciIoWidthUint64`**, then both *`Address`* and *`Buffer`* are incremented for each of the *`Count`* operations performed.

If *`Width`* is **`EfiPciIoWidthFifoUint8`**, **`EfiPciIoWidthFifoUint16`**, **`EfiPciIoWidthFifoUint32`**, or **`EfiPciIoWidthFifoUint64`**, then only *`Buffer`* is incremented for each of the *`Count`* operations performed. The read or write operation is performed *`Count`* times on the same *`Address`*.

If *`Width`* is **`EfiPciIoWidthFillUint8`**, **`EfiPciIoWidthFillUint16`**, **`EfiPciIoWidthFillUint32`**, or **`EfiPciIoWidthFillUint64`**, then only *`Address`* is incremented for each of the *`Count`* operations performed. The read or write operation is performed *`Count`* times from the first element of *`Buffer`*.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PCI controller.
EFI_INVALID_PARAMETER	*`Width`* is invalid.
EFI_INVALID_PARAMETER	*`Buffer`* is **`NULL`**.
EFI_UNSUPPORTED	The address range specified by *`Offset`*, *`Width`*, and *`Count`* is not valid for the PCI configuration header of the PCI controller.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_IO_PROTOCOL.CopyMem()

Summary

Enables a PCI driver to copy one region of PCI memory space to another region of PCI memory space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_COPY_MEM) (
  IN     EFI_PCI_IO_PROTOCOL         *This,
  IN     EFI_PCI_IO_PROTOCOL_WIDTH   Width,
  IN     UINT8                       DestBarIndex,
  IN     UINT64                      DestOffset,
  IN     UINT8                       SrcBarIndex,
  IN     UINT64                      SrcOffset,
  IN     UINTN                       Count
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Width	Signifies the width of the memory operations. Type **EFI_PCI_IO_PROTOCOL_WIDTH** is defined in Section 12.4.
DestBarIndex	The BAR index in the standard PCI Configuration header to use as the base address for the memory operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value **EFI_PCI_IO_PASS_THROUGH_BAR** can be used to bypass the BAR relative addressing and pass *Offset* to the PCI Root Bridge I/O Protocol unchanged. Type **EFI_PCI_IO_PASS_THROUGH_BAR** is defined in Section 12.4.
DestOffset	The destination offset within the BAR specified by *DestBarIndex* to start the memory writes for the copy operation. The caller is responsible for aligning the *DestOffset* if required.

SrcBarIndex	The BAR index in the standard PCI Configuration header to use as the base address for the memory operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value **EFI_PCI_IO_PASS_THROUGH_BAR** can be used to bypass the BAR relative addressing and pass *Offset* to the PCI Root Bridge I/O Protocol unchanged. Type **EFI_PCI_IO_PASS_THROUGH_BAR** is defined in Section 12.4.
SrcOffset	The source offset within the BAR specified by *SrcBarIndex* to start the memory reads for the copy operation. The caller is responsible for aligning the *SrcOffset* if required.
Count	The number of memory operations to perform. Bytes moved is *Width* size * *Count*, starting at *DestOffset* and *SrcOffset*.

Description

The **CopyMem()** function enables a PCI driver to copy one region of PCI memory space to another region of PCI memory space on a PCI controller. This is especially useful for video scroll operations on a memory mapped video buffer.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI controller on a platform might require. For example on some platforms, width requests of **EfiPciIoWidthUint64** do not work.

If *Width* is **EfiPciWidthUint8**, **EfiPciWidthUint16**, **EfiPciWidthUint32**, or **EfiPciWidthUint64**, then *Count* read/write transactions are performed to move the contents of the *SrcOffset* buffer to the *DestOffset* buffer. The implementation must be reentrant, and it must handle overlapping *SrcOffset* and *DestOffset* buffers. This means that the implementation of **CopyMem()** must choose the correct direction of the copy operation based on the type of overlap that exists between the *SrcOffset* and *DestOffset* buffers. If either the *SrcOffset* buffer or the *DestOffset* buffer crosses the top of the processor's address space, then the result of the copy operation is unpredictable.

The contents of the *DestOffset* buffer on exit from this service must match the contents of the *SrcOffset* buffer on entry to this service. Due to potential overlaps, the contents of the *SrcOffset* buffer may be modified by this service. The following rules can be used to guarantee the correct behavior:

1. If *DestOffset* > *SrcOffset* **and** *DestOffset* < (*SrcOffset* + *Width* size * *Count*), then the data should be copied from the *SrcOffset* buffer to the *DestOffset* buffer starting from the end of buffers and working toward the beginning of the buffers.
2. Otherwise, the data should be copied from the *SrcOffset* buffer to the *DestOffset* buffer starting from the beginning of the buffers and working toward the end of the buffers.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the *PCI Specification*. However, if the memory-mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Status Codes Returned

EFI_SUCCESS	The data was copied from one memory region to another memory region.
EFI_INVALID_PARAMETER	*Width* is invalid.
EFI_UNSUPPORTED	*DestBarIndex* not valid for this PCI controller.
EFI_UNSUPPORTED	*SrcBarIndex* not valid for this PCI controller.
EFI_UNSUPPORTED	The address range specified by *DestOffset*, *Width*, and *Count* is not valid for the PCI BAR specified by *DestBarIndex*.
EFI_UNSUPPORTED	The address range specified by *SrcOffset*, *Width*, and *Count* is not valid for the PCI BAR specified by *SrcBarIndex*.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_IO_PROTOCOL.Map()

Summary

Provides the PCI controller–specific addresses needed to access system memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_MAP) (
  IN      EFI_PCI_IO_PROTOCOL             *This,
  IN      EFI_PCI_IO_PROTOCOL_OPERATION   Operation,
  IN      VOID                            *HostAddress,
  IN OUT  UINTN                           *NumberOfBytes,
  OUT     EFI_PHYSICAL_ADDRESS            *DeviceAddress,
  OUT     VOID                            **Mapping
);
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Operation	Indicates if the bus master is going to read or write to system memory. Type **EFI_PCI_IO_PROTOCOL_OPERATION** is defined in Section 12.4.
HostAddress	The system memory address to map to the PCI controller.
NumberOfBytes	On input the number of bytes to map. On output the number of bytes that were mapped.
DeviceAddress	The resulting map address for the bus master PCI controller to use to access the hosts *HostAddress*. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5. This address cannot be used by the processor to access the contents of the buffer specified by *HostAddress*.
Mapping	A resulting value to pass to **Unmap()**.

Description

The **Map()** function provides the PCI controller–specific addresses needed to access system memory. This function is used to map system memory for PCI bus master DMA accesses.

All PCI bus master accesses must be performed through their mapped addresses and such mappings must be freed with **Unmap()** when complete. If the bus master access is a single read or write data transfer, then **EfiPciIoOperationBusMasterRead** or **EfiPciIoOperation-BusMasterWrite** is used and the range is unmapped to complete the operation. If performing an **EfiPciIoOperationBusMasterRead** operation, all the data must be present in system memory before the **Map()** is performed. Similarly, if performing an **EfiPciIoOperation-BusMasterWrite,** the data cannot be properly accessed in system memory until **Unmap()** is performed.

Bus master operations that require both read and write access or require multiple host device interactions within the same mapped region must use **EfiPciIoOperation-BusMasterCommonBuffer**. However, only memory allocated via the **AllocateBuffer()** interface can be mapped for this operation type.

In all mapping requests the resulting *NumberOfBytes* actually mapped may be less than the requested amount. In this case, the DMA operation will have to be broken up into smaller chunks. The **Map()** function will map as much of the DMA operation as it can at one time. The caller may have to loop on **Map()** and **Unmap()** in order to complete a large DMA transfer.

Status Codes Returned

EFI_SUCCESS	The range was mapped for the returned *NumberOfBytes*.
EFI_INVALID_PARAMETER	*Operation* is invalid.
EFI_INVALID_PARAMETER	*HostAddress* is **NULL**.
EFI_INVALID_PARAMETER	*NumberOfBytes* is **NULL**.
EFI_INVALID_PARAMETER	*DeviceAddress* is **NULL**.
EFI_INVALID_PARAMETER	*Mapping* is **NULL**.
EFI_UNSUPPORTED	The *HostAddress* cannot be mapped as a common buffer.
EFI_DEVICE_ERROR	The system hardware could not map the requested address.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_IO_PROTOCOL.Unmap()

Summary

Completes the **Map()** operation and releases any corresponding resources.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_UNMAP) (
  IN   EFI_PCI_IO_PROTOCOL    *This,
  IN   VOID                   *Mapping
);
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Mapping	The mapping value returned from **Map()**.

Description

The **Unmap()** function completes the **Map()** operation and releases any corresponding resources. If the operation was an **EfiPciIoOperationBusMasterWrite**, the data is committed to the target system memory. Any resources used for the mapping are freed.

Status Codes Returned

EFI_SUCCESS	The range was unmapped.
EFI_INVALID_PARAMETER	*HostAddress* is not a value that was returned by **Map()**.
EFI_DEVICE_ERROR	The data was not committed to the target system memory.

EFI_PCI_IO_PROTOCOL.AllocateBuffer()

Summary

Allocates pages that are suitable for an **EfiPciIoOperationBusMasterCommonBuffer** mapping.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_ALLOCATE_BUFFER) (
  IN      EFI_PCI_IO_PROTOCOL    *This,
  IN      EFI_ALLOCATE_TYPE      Type,
  IN      EFI_MEMORY_TYPE        MemoryType,
  IN      UINTN                  Pages,
  OUT     VOID                   **HostAddress,
  IN      UINT64                 Attributes
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Type	This parameter is not used and must be ignored.
MemoryType	The type of memory to allocate, **EfiBootServicesData** or **EfiRuntimeServicesData**. Type **EFI_MEMORY_TYPE** is defined in Chapter 5.
Pages	The number of pages to allocate.
HostAddress	A pointer to store the base system memory address of the allocated range.
Attributes	The requested bit mask of attributes for the allocated range. Only the attributes **EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE**, and **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** may be used with this function. If any other bits are set, then **EFI_UNSUPPORTED** is returned. This function may choose to ignore this bit mask. The **EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE**, and **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attributes provide a hint to the implementation that may improve the performance of the calling driver. The implementation may choose any default for the memory attributes including write combining, cached, both, or neither as long as the allocated buffer can be seen equally by both the processor and the PCI bus master.

Description

The **AllocateBuffer()** function allocates pages that are suitable for an **EfiPciIoOperationBusMasterCommonBuffer** mapping. This means that the buffer allocated by this function must support simultaneous access by both the processor and a PCI Bus Master. The device address that the PCI Bus Master uses to access the buffer can be retrieved with a call to **Map()**.

If the current attributes of the PCI controller has the **EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE** bit set, then when the buffer allocated by this function is mapped with a call to **Map()**, the device address that is returned by **Map()** must be within the 64-bit device address space of the PCI Bus Master. The attributes for a PCI controller can be managed by calling **Attributes()**.

If the current attributes for the PCI controller has the **EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE** bit clear, then when the buffer allocated by this function is mapped with a call to **Map()**, the device address that is returned by **Map()** must be within the 32-bit device address space of the PCI Bus Master. The attributes for a PCI controller can be managed by calling **Attributes()**.

If the memory allocation specified by *MemoryType* and *Pages* cannot be satisfied, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were allocated.
EFI_INVALID_PARAMETER	*MemoryType* is invalid.
EFI_INVALID_PARAMETER	*HostAddress* is **NULL**.
EFI_UNSUPPORTED	*Attributes* is unsupported. The only legal attribute bits are **MEMORY_WRITE_COMBINE** and **MEMORY_CACHED**.
EFI_OUT_OF_RESOURCES	The memory pages could not be allocated.

EFI_PCI_IO_PROTOCOL.FreeBuffer()

Summary

Frees memory that was allocated with **AllocateBuffer()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_FREE_BUFFER) (
  IN  EFI_PCI_IO_PROTOCOL    *This,
  IN  UINTN                  Pages,
  IN  VOID                   *HostAddress
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Pages	The number of pages to free.
HostAddress	The base system memory address of the allocated range.

Description

The **FreeBuffer()** function frees memory that was allocated with **AllocateBuffer()**.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were freed.
EFI_INVALID_PARAMETER	The memory range specified by *HostAddress* and *Pages* was not allocated with **AllocateBuffer()**.

EFI_PCI_IO_PROTOCOL.Flush()

Summary

Flushes all PCI posted write transactions from a PCI host bridge to system memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_FLUSH) (
  IN  EFI_PCI_IO_PROTOCOL   *This
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.

Description

The **Flush()** function flushes any PCI posted write transactions from a PCI host bridge to system memory. Posted write transactions are generated by PCI bus masters when they perform write transactions to target addresses in system memory.

This function does not flush posted write transactions from any PCI bridges. A PCI controller specific action must be taken to guarantee that the posted write transactions have been flushed from the PCI controller and from all the PCI bridges into the PCI host bridge. This is typically done with a PCI read transaction from the PCI controller prior to calling **Flush()**.

If the PCI controller specific action required to flush the PCI posted write transactions has been performed, and this function returns **EFI_SUCCESS**, then the PCI bus master's view and the processor's view of system memory are guaranteed to be coherent. If the PCI posted write transactions cannot be flushed from the PCI host bridge, then the PCI bus master and processor are not guaranteed to have a coherent view of system memory, and **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	The PCI posted write transactions were flushed from the PCI host bridge to system memory.
EFI_DEVICE_ERROR	The PCI posted write transactions were not flushed from the PCI host bridge due to a hardware error.

EFI_PCI_IO_PROTOCOL.GetLocation()

Summary

Retrieves this PCI controller's current PCI bus number, device number, and function number.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_GET_LOCATION) (
  IN  EFI_PCI_IO_PROTOCOL   *This,
  OUT UINTN                 *SegmentNumber,
  OUT UINTN                 *BusNumber,
  OUT UINTN                 *DeviceNumber,
  OUT UINTN                 *FunctionNumber
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
SegmentNumber	The PCI controller's current PCI segment number.
BusNumber	The PCI controller's current PCI bus number.
DeviceNumber	The PCI controller's current PCI device number.
FunctionNumber	The PCI controller's current PCI function number.

Description

The **GetLocation()** function retrieves a PCI controller's current location on a PCI Host Bridge. This is specified by a PCI segment number, PCI bus number, PCI device number, and PCI function number. These values can be used with the PCI Root Bridge I/O Protocol to perform PCI configuration cycles on the PCI controller, or any of its peer PCI controller's on the same PCI Host Bridge.

Status Codes Returned

EFI_SUCCESS	The PCI controller location was returned.
EFI_INVALID_PARAMETER	*SegmentNumber* is **NULL**.
EFI_INVALID_PARAMETER	*BusNumber* is **NULL**.
EFI_INVALID_PARAMETER	*DeviceNumber* is **NULL**.
EFI_INVALID_PARAMETER	*FunctionNumber* is **NULL**.

EFI_PCI_IO_PROTOCOL.Attributes()

Summary

Performs an operation on the attributes that this PCI controller supports. The operations include getting the set of supported attributes, retrieving the current attributes, setting the current attributes, enabling attributes, and disabling attributes.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_ATTRIBUTES) (
  IN  EFI_PCI_IO_PROTOCOL                          *This,
  IN  EFI_PCI_IO_PROTOCOL_ATTRIBUTE_OPERATION      Operation,
  IN  UINT64                                       Attributes,
  OUT UINT64                                       *Result    OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Operation	The operation to perform on the attributes for this PCI controller. Type **EFI_PCI_IO_PROTOCOL_ATTRIBUTE_OPERATION** is defined in “Related Definitions” below.
Attributes	The mask of attributes that are used for **Set**, **Enable**, and **Disable** operations. The available attributes are listed in Section 12.4.
Result	A pointer to the result mask of attributes that are returned for the **Get** and **Supported** operations. This is an optional parameter that may be **NULL** for the **Set**, **Enable**, and **Disable** operations. The available attributes are listed in Section 12.4.

Related Definitions

```
//*******************************************************
// EFI_PCI_IO_PROTOCOL_ATTRIBUTE_OPERATION
//*******************************************************
typedef enum {
  EfiPciIoAttributeOperationGet,
  EfiPciIoAttributeOperationSet,
  EfiPciIoAttributeOperationEnable,
  EfiPciIoAttributeOperationDisable,
  EfiPciIoAttributeOperationSupported,
  EfiPciIoAttributeOperationMaximum
} EFI_PCI_IO_PROTOCOL_ATTRIBUTE_OPERATION;
```

EfiPciIoAttributeOperationGet

Retrieve the PCI controller's current attributes, and return them in *Result*. If *Result* is **NULL**, then **EFI_INVALID_PARAMER** is returned. For this operation, *Attributes* is ignored.

EfiPciIoAttributeOperationSet

Set the PCI controller's current attributes to *Attributes*. If a bit is set in *Attributes* that is not supported by this PCI controller or one of its parent bridges, then **EFI_UNSUPPORTED** is returned. For this operation, *Result* is an optional parameter that may be **NULL**.

EfiPciIoAttributeOperationEnable

Enable the attributes specified by the bits that are set in *Attributes* for this PCI controller. Bits in *Attributes* that are clear are ignored. If a bit is set in *Attributes* that is not supported by this PCI controller or one of its parent bridges, then **EFI_UNSUPPORTED** is returned. For this operation, *Result* is an optional parameter that may be **NULL**.

EfiPciIoAttributeOperationDisable

Disable the attributes specified by the bits that are set in *Attributes* for this PCI controller. Bits in *Attributes* that are clear are ignored. If a bit is set in *Attributes* that is not supported by this PCI controller or one of its parent bridges, then **EFI_UNSUPPORTED** is returned. For this operation, *Result* is an optional parameter that may be **NULL**.

EfiPciIoAttributeOperationSupported

Retrieve the PCI controller's supported attributes, and return them in *Result*. If *Result* is **NULL**, then **EFI_INVALID_PARAMER** is returned. For this operation, *Attributes* is ignored.

Description

The **`Attributes()`** function performs an operation on the attributes associated with this PCI controller. If *`Operation`* is greater than or equal to the maximum operation value, then **`EFI_INVALID_PARAMETER`** is returned. If *`Operation`* is **`Get`** or **`Supported`**, and *`Result`* is **`NULL`**, then **`EFI_INVALID_PARAMETER`** is returned. If *`Operation`* is **`Set`**, **`Enable`**, or **`Disable`** for an attribute that is not supported by the PCI controller, then **`EFI_UNSUPPORTED`** is returned. Otherwise, the operation is performed as described in "Related Definitions" and **`EFI_SUCCESS`** is returned. It is possible for this function to return **`EFI_UNSUPPORTED`** even if the PCI controller supports the attribute. This can occur when the PCI root bridge does not support the attribute. For example, if VGA I/O and VGA Memory transactions cannot be forwarded onto PCI root bridge #2, then a request by a PCI VGA driver to enable the **`VGA_IO`** and **`VGA_MEMORY`** bits will fail even though a PCI VGA controller behind PCI root bridge #2 is able to decode these transactions.

This function will also return **`EFI_UNSUPPORTED`** if more than one PCI controller on the same PCI root bridge has already successfully requested one of the ISA addressing attributes. For example, if one PCI VGA controller had already requested the **`VGA_IO`** and **`VGA_MEMORY`** attributes, then a second PCI VGA controller on the same root bridge cannot succeed in requesting those same attributes. This restriction applies to the ISA-, VGA-, and IDE-related attributes.

Status Codes Returned

EFI_SUCCESS	The operation on the PCI controller's attributes was completed. If the operation was **`Get`** or **`Supported`**, then the attribute mask is returned in *`Result`*.
EFI_INVALID_PARAMETER	*`Operation`* is greater than or equal to **`EfiPciIoAttributeOperationMaximum`**.
EFI_INVALID_PARAMETER	*`Operation`* is **`Get`** and *`Result`* is **`NULL`**.
EFI_INVALID_PARAMETER	*`Operation`* is **`Supported`** and *`Result`* is **`NULL`**.
EFI_UNSUPPORTED	*`Operation`* is **`Set`**, and one or more of the bits set in *`Attributes`* are not supported by this PCI controller or one of its parent bridges.
EFI_UNSUPPORTED	*`Operation`* is **`Enable`**, and one or more of the bits set in *`Attributes`* are not supported by this PCI controller or one of its parent bridges.
EFI_UNSUPPORTED	*`Operation`* is **`Disable`**, and one or more of the bits set in *`Attributes`* are not supported by this PCI controller or one of its parent bridges.

EFI_PCI_IO_PROTOCOL.GetBarAttributes()

Summary

Gets the attributes that this PCI controller supports setting on a BAR using **SetBarAttributes()**, and retrieves the list of resource descriptors for a BAR.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_GET_BAR_ATTRIBUTES) (
  IN  EFI_PCI_IO_PROTOCOL   *This,
  IN  UINT8                 BarIndex,
  OUT UINT64                *Supports    OPTIONAL,
  OUT VOID                  **Resources  OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
BarIndex	The BAR index of the standard PCI Configuration header to use as the base address for resource range. The legal range for this field is 0..5.
Supports	A pointer to the mask of attributes that this PCI controller supports setting for this BAR with **SetBarAttributes()**. The list of attributes is listed in Section 12.4. This is an optional parameter that may be **NULL**.
Resources	A pointer to the ACPI 2.0 resource descriptors that describe the current configuration of this BAR of the PCI controller. This buffer is allocated for the caller with the Boot Service **AllocatePool()**. It is the caller's responsibility to free the buffer with the Boot Service **FreePool()**. See "Related Definitions" below for the ACPI 2.0 resource descriptors that may be used. This is an optional parameter that may be **NULL**.

Related Definitions

There are only two resource descriptor types from the *ACPI Specification* that may be used to describe the current resources allocated to BAR of a PCI Controller. These are the QWORD Address Space Descriptor (ACPI 2.0 Section 6.4.3.5.1), and the End Tag (ACPI 2.0 Section 6.4.2.8). The QWORD Address Space Descriptor can describe memory, I/O, and bus number ranges for dynamic or fixed resources. The configuration of a BAR of a PCI Controller is described with one or more QWORD Address Space Descriptors followed by an End Tag. Table 12-10 and Table 12-11 contain these two descriptor types. Please see the *ACPI Specification* for details on the field values.

Table 12-10. ACPI 2.0 QWORD Address Space Descriptor

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x8A	QWORD Address Space Descriptor
0x01	0x02	0x2B	Length of this descriptor in bytes not including the first two fields
0x03	0x01		Resource Type 0 – Memory Range 1 – I/O Range 2 – Bus Number Range
0x04	0x01		General Flags
0x05	0x01		Type Specific Flags
0x06	0x08		Address Space Granularity
0x0E	0x08		Address Range Minimum
0x16	0x08		Address Range Maximum
0x1E	0x08		Address Translation Offset
0x26	0x08		Address Length

Table 12-11. ACPI 2.0 End Tag

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x79	End Tag
0x01	0x01	0x00	Checksum. If 0, then checksum is assumed to be valid.

Description

The **GetBarAttributes()** function returns in *Supports* the mask of attributes that the PCI controller supports setting for the BAR specified by *BarIndex*. It also returns in *Resources* a list of ACPI 2.0 resource descriptors for the BAR specified by *BarIndex*. Both *Supports* and *Resources* are optional parameters. If both *Supports* and *Resources* are **NULL**, then **EFI_INVALID_PARAMETER** is returned. It is the caller's responsibility to free *Resources* with the Boot Service **FreePool()** when the caller is done with the contents of *Resources*. If there are not enough resources to allocate *Resources*, then **EFI_OUT_OF_RESOURCES** is returned.

If a bit is set in *Supports*, then the PCI controller supports this attribute type for the BAR specified by *BarIndex*, and a call can be made to **SetBarAttributes()** using that attribute type.

Status Codes Returned

EFI_SUCCESS	If *Supports* is not **NULL**, then the attributes that the PCI controller supports are returned in *Supports*. If *Resources* is not **NULL**, then the ACPI 2.0 resource descriptors that the PCI controller is currently using are returned in *Resources*.
EFI_OUT_OF_RESOURCES	There are not enough resources available to allocate *Resources*.
EFI_UNSUPPORTED	*BarIndex* not valid for this PCI controller.
EFI_INVALID_PARAMETER	Both *Supports* and *Attributes* are **NULL**.

EFI_PCI_IO_PROTOCOL.SetBarAttributes()

Summary

Sets the attributes for a range of a BAR on a PCI controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_SET_BAR_ATTRIBUTES) (
  IN     EFI_PCI_IO_PROTOCOL   *This,
  IN     UINT64                Attributes,
  IN     UINT8                 BarIndex,
  IN OUT UINT64                *Offset,
  IN OUT UINT64                *Length
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Attributes	The mask of attributes to set for the resource range specified by *BarIndex*, *Offset*, and *Length*.
BarIndex	The BAR index of the standard PCI Configuration header to use as the base address for the resource range. The legal range for this field is 0..5.
Offset	A pointer to the BAR relative base address of the resource range to be modified by the attributes specified by *Attributes*. On return, **Offset* will be set to the actual base address of the resource range. Not all resources can be set to a byte boundary, so the actual base address may differ from the one passed in by the caller.
Length	A pointer to the length of the resource range to be modified by the attributes specified by *Attributes*. On return, **Length* will be set to the actual length of the resource range. Not all resources can be set to a byte boundary, so the actual length may differ from the one passed in by the caller.

Description

The **SetBarAttributes()** function sets the attributes specified in *Attributes* for the PCI controller on the resource range specified by *BarIndex*, *Offset*, and *Length*. Since the granularity of setting these attributes may vary from resource type to resource type, and from platform to platform, the actual resource range and the one passed in by the caller may differ. As a result, this function may set the attributes specified by *Attributes* on a larger resource range than the caller requested. The actual range is returned in *Offset* and *Length*. The caller is responsible for verifying that the actual range for which the attributes were set is acceptable.

If the attributes are set on the PCI controller, then the actual resource range is returned in *Offset* and *Length*, and **EFI_SUCCESS** is returned. Many of the attribute types also require that the state of the PCI Host Bus Controller and the state of any PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller to be modified. This function will only return **EFI_SUCCESS** is all of these state changes are made. The PCI Controller may support a combination of attributes, but unless the PCI Host Bus Controller and the PCI to PCI bridges also support that same combination of attributes, then this call will return an error.

If the attributes specified by *Attributes*, or the resource range specified by *BarIndex*, *Offset*, and *Length* are not supported by the PCI controller, then **EFI_UNSUPPORTED** is returned. The set of supported attributes for the PCI controller can be found by calling **GetBarAttributes()**.

If either *Offset* or *Length* is **NULL** then **EFI_INVALID_PARAMETER** is returned.

If there are not enough resources available to set the attributes, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The set of attributes specified by *Attributes* for the resource range specified by *BarIndex*, *Offset*, and *Length* were set on the PCI controller, and the actual resource range is returned in *Offset* and *Length*.
EFI_UNSUPPORTED	The set of attributes specified by *Attributes* is not supported by the PCI controller for the resource range specified by *BarIndex*, *Offset*, and *Length*.
EFI_INVALID_PARAMETER	*Offset* is **NULL**.
EFI_INVALID_PARAMETER	*Length* is **NULL**.
EFI_OUT_OF_RESOURCES	There are not enough resources to set the attributes on the resource range specified by *BarIndex*, *Offset*, and *Length*.

12.4.1 PCI Device Paths

An **EFI_PCI_IO_PROTOCOL** must be installed on a handle for its services to be available to PCI device drivers. In addition to the **EFI_PCI_IO_PROTOCOL**, an **EFI_DEVICE_PATH** must also be installed on the same handle. See Chapter 5 for a detailed description of the **EFI_DEVICE_PATH**.

Typically, an ACPI Device Path Node is used to describe a PCI Root Bridge. Depending on the bus hierarchy in the system, additional device path nodes may precede this ACPI Device Path Node. A PCI device path is described with PCI Device Path Nodes. There will be one PCI Device Path node for the PCI controller itself, and one PCI Device Path Node for each PCI to PCI Bridge that is between the PCI controller and the PCI Root Bridge.

Table 12-12 shows an example device path for a PCI controller that is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. This device path consists of an ACPI Device Path Node, a PCI Device Path Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(7|0).
```

Table 12-12. PCI Device 7, Function 0 on PCI Root Bridge 0

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x00	PCI Function
0x11	0x01	0x07	PCI Device
0x12	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x13	0x01	0xFF	Sub type – End of Entire Device Path
0x14	0x02	0x04	Length – 0x04 bytes

Table 12-13 shows an example device path for a PCI controller that is located behind a PCI to PCI bridge at PCI device number 0x07 and PCI function 0x00. The PCI to PCI bridge is directly attached to a PCI root bridge, and it is at PCI device number 0x05 and PCI function 0x00. This device path consists of an ACPI Device Path Node, two PCI Device Path Nodes, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(5|0)/PCI(7|0).
```

Table 12-13. PCI Device 7, Function 0 behind PCI to PCI bridge

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes.
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x00	PCI Function
0x11	0x01	0x05	PCI Device
0x12	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x13	0x01	0x01	Sub type – PCI
0x14	0x02	0x06	Length – 0x06 bytes
0x16	0x01	0x00	PCI Function
0x17	0x01	0x07	PCI Device
0x18	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x19	0x01	0xFF	Sub type – End of Entire Device Path
0x1A	0x02	0x04	Length – 0x04 bytes

12.4.2 PCI Option ROMs

EFI takes advantage of both the *PCI Specification* and the *PE/COFF Specification* to store EFI images in a PCI Option ROM. There are several rules that must be followed when constructing a PCI Option ROM

- A PCI Option ROM can be no larger than 16 MB.
- A PCI Option ROM may contain one or more images.
- Each image must being on a 512-byte boundary.
- Each image must be an even multiple of 512 bytes in length. This means that images that are not an even multiple of 512 bytes in length must be padded to the next 512-byte boundary.
- Legacy Option ROM images begin with a Standard PCI Expansion ROM Header (Table 12-14).
- EFI Option ROM images begin with an EFI PCI Expansion ROM Header (Table 12-17).
- Each image must contain a PCIR data structure in the first 64 KB of the image (Table 12-15).
- The image data for an EFI Option ROM image must begin in the first 64 KB of the image.
- The image data for an EFI Option ROM image must be a PE/COFF image or a compressed PE/COFF image following the *EFI 1.10 Compression Algorithm Specification*.
- The PCIR data structure must being in a 4-byte boundary.
- If the PCI Option ROM contains a Legacy Option ROM image, it must be the first image.
- The images are placed in the PCI Option ROM is order from highest to lowest priority. This priority is used to build the ordered list of Driver Image Handles that are produced by the Bus Specific Driver Override Protocol for a PCI Controller.

There are several options available when building a PCI option ROM for a PCI adapter. A PCI Option ROM can choose to support only a legacy PC-AT platform, only an EFI compliant platform, or both. This flexibility allows a migration path from adapters that support only legacy PC-AT platforms, to adapters that support both PC-AT platforms and EFI compliant platforms, to adapters that support only EFI compliant platforms. The following is a list of the image combinations that may be placed in a PCI option ROM. This is not an exhaustive list. Instead, it provides what will likely be the most common PCI option ROM layouts. EFI complaint system firmware must work with all of these PCI option ROM layouts, plus any other layouts that are possible within the *PCI Specification*. The format of a Legacy Option ROM image is defined in the *PCI Specification*.

- Legacy Option ROM image
- Legacy Option ROM image + IA-32 EFI Driver
- Legacy Option ROM image + Itanium Processor Family EFI Driver
- Legacy Option ROM image + IA-32 EFI Driver + Itanium Processor Family EFI Driver
- Legacy Option ROM image + EBC Driver
- IA-32 EFI Driver
- Itanium Processor Family EFI Driver
- IA-32 EFI Driver + Itanium Processor Family EFI Driver
- EBC Driver

It is also possible to place an EFI Application in a PCI Option ROM. However, the PCI Bus Driver will ignore these images. The exact mechanism by which EFI Applications can be loaded and executed from a PCI Option ROM is outside the scope of this document.

Table 12-14. Standard PCI Expansion ROM Header

Offset	Byte Length	Value	Description
0x00	1	0x55	ROM Signature, byte 1
0x01	1	0xAA	ROM Signature, byte 2
0x02-0x17	22	XX	Reserved per processor architecture unique data
0x18-0x19	2	XX	Pointer to PCIR Data Structure

Table 12-15. PCIR Data Structure

Offset	Byte Length	Description
0x00	4	Signature, the string 'PCIR'
0x04	2	Vendor Identification
0x06	2	Device Identification
0x08	2	Pointer to Vital Product Data
0x0a	2	PCIR Data Structure Length
0x0c	1	PCIR Data Structure Revision
0x0d	3	Class Code
0x10	2	Image Length
0x12	2	Revision Level of Code/Data
0x14	1	Code Type
0x15	1	Indicator. Used to identify if this is the last image in the ROM
0x16	2	Reserved

Table 12-16. PCI Expansion ROM Code Types

Code Type	Description
0x00	Intel® IA-32, PC-AT compatible
0x01	Open Firmware standard for PCI
0x02	Hewlett-Packard PA RISC
0x03	EFI Image
0x04-0xFF	Reserved

Table 12-17. EFI PCI Expansion ROM Header

Offset	Byte Length	Value	Description
0x00	1	0x55	ROM Signature, byte 1
0x01	1	0xAA	ROM Signature, byte 2
0x02	2	XXXX	Initialization Size – size of this image in units of 512 bytes. The size includes this header.
0x04	4	0x0EF1	Signature from EFI image header
0x08	2	XX	Subsystem value for EFI image header
0x0a	2	XX	Machine type from EFI image header
0x0c	2	XX	Compression type 0x0000 - The image is uncompressed 0x0001 - The image is compressed. See the *EFI 1.1 Compression Algorithm Specification*. 0x0002 - 0xFFFF - Reserved
0x0e	8	0x00	Reserved
0x16	2	XX	Offset to EFI Image
0x18	2	XX	Offset to PCIR Data Structure

12.4.2.1 PCI Bus Driver Responsibilities

A PCI Bus Driver must scan a PCI Option ROM for PCI Device Drivers. If a PCI Option ROM is found during PCI Enumeration, then a copy of the PCI Option ROM is placed in a memory buffer. The PCI Bus Driver will use the memory copy of the PCI Option ROM to search for EFI Drivers after PCI Enumeration. The PCI Bus Driver will search the list of images in a PCI Option ROM for the ones that have a Code Type of 0x03 in the PCIR Data Structure, and a Signature of 0xEF1 in the EFI PCI Expansion ROM Header. Then, it will examine the Subsystem Type of the EFI PCI Expansion ROM Header. If the Subsystem Type is **IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER**(11) or **IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER**(12), then the PCI Bus Driver can load the PCI Device Driver from the PCI Option ROM. The Offset to EFI Image Header field of the EFI PCI Expansion ROM Header is used to get a pointer to the beginning of the PE/COFF image in the PCI Option ROM. The PE/COFF image may have been compressed using the *EFI 1.10* Compression Algorithm. If it has been compressed, then the PCI Bus Driver must decompress the driver to a memory buffer. The Boot Service **LoadImage()** can then be used to load the EFI Driver. If the platform does not support the Machine Type of the EFI Driver, then **LoadImage()** may fail. For example, an EFI Driver with an Itanium processor of type of 0x200 would fail to load on an IA-32 platform. If **LoadImage()** succeeds, then the Boot Service **StartImage()** can be called to start the PCI Device Driver. The Image Length field of the PCIR Data Structure can be used to get to the next image in the PCI Option ROM. The PCI Option ROM search is completed when an image is found whose Indicator field of the PCIR Data Structure has bit 7 set.

It is the PCI Bus Driver's responsibility to verify that the Expansion ROM Header and PCIR Data Structure are valid. It is the responsibly of the Boot Service `LoadImage()` to verify that the PE/COFF image is valid. The Boot Service `LoadImage()` may fail for several reasons including a corrupt PE/COFF image or an unsupported Machine Type.

The PCI Option ROM search may produce one or more Driver Image Handles for the PCI Controller that is associated with the PCI Option ROM. The PCI Bus Driver is responsible for producing a Bus Specific Driver Override Protocol instance for every PCI Controller has a PCI Option ROM that contains one or more EFI Drivers. The Bus Specific Driver Override Protocol produces an ordered list of Driver Image Handles. The order that the EFI Drivers are placed in the PCI Option ROM is the order of Driver Image Handles that must be returned by the Bus Specific Driver Override Protocol. This gives the party that builds the PCI Option ROM control over the order that the EFI Drivers are used in the Boot Service `ConnectController()`.

12.4.2.2 PCI Device Driver Responsibilities

A PCI Device Driver should not be designed to care where it is stored. It can reside in a PCI Option ROM, the system's motherboard ROM, a hard drive, a CD-ROM drive, etc. All PCI Device Drivers are compiled and linked to generate a PE/COFF image. When a PE/COFF image is placed in a PCI Option ROM, it must follow the rules outlined in Section 2.7. The recommended image layout is to insert an EFI PCI Expansion ROM Header and a PCIR Data Structure in front of the PE/COFF image, and pad the entire image up to the next 512-byte boundary. Figure 12-15 shows the format of a single PCI Device Driver that can be added to a PCI Option ROM.

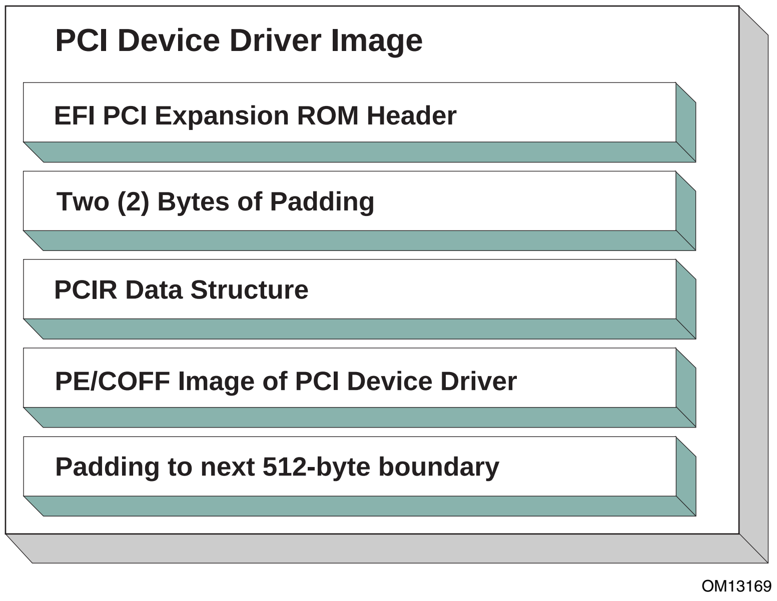

PCI Device Driver Image
EFI PCI Expansion ROM Header
Two (2) Bytes of Padding
PCIR Data Structure
PE/COFF Image of PCI Device Driver
Padding to next 512-byte boundary
OM13169

Figure 12-15. Recommended PCI Driver Image Layout

The field values for the EFI PCI Expansion ROM Header and the PCIR Data Structure would be as follows in this recommended PCI Driver image layout. An image must start at a 512-byte boundary, and the end of the image must be padded to the next 512-byte boundary.

Table 12-18. Recommended PCI Device Driver Layout

Offset	Byte Length	Value	Description
0x00	1	0x55	ROM Signature, byte 1
0x01	1	0xAA	ROM Signature, byte 2
0x02	2	XXXX	Initialization Size – size of this image in units of 512 bytes. The size includes this header
0x04	4	0x0EF1	Signature from EFI image header
0x08	2	XX 0x0B 0x0C	Subsystem Value from the PCI Driver's PE/COFF Image Header Subsystem Value for an EFI Boot Service Driver Subsystem Value for an EFI Runtime Driver
0x0a	2	XX 0x014C 0x0200 0x0EBC	Machine type from the PCI Driver's PE/COFF Image Header IA-32 Machine Type Itanium processor type EFI Byte Code (EBC) Machine Type
0x0C	2	XXXX 0x0000 0x0001	Compression Type Uncompressed Compressed following the *EFI 1.10 Compression Algorithm Specification*
0x0E	8	0x00	Reserved
0x16	2	0x0034	Offset to EFI Image
0x18	2	0x001C	Offset to PCIR Data Structure
0x1A	2	0x0000	Padding to align PCIR Data Structure on a 4 byte boundary
0x1C	4	'PCIR'	PCIR Data Structure Signature
0x20	2	XXXX	Vendor ID from the PCI Controller's Configuration Header
0x22	2	XXXX	Device ID from the PCI Controller's Configuration Header
0x24	2	0x0000	Reserved
0x26	2	0x0018	The length if the PCIR Data Structure in bytes
0x28	1	0x00	PCIR Data Structure Revision. Value for PCI 2.2 Option ROM
0x29	3	XXXX	Class Code from the PCI Controller's Configuration Header
0x2C	2	XXXX	Code Image Length in units of 512 bytes. Same as Initialization Size
0x2E	2	XXXX	Revision Level of the Code/Data. This field is ignored

continued

Table 12-18. Recommended PCI Device Driver Layout (continued)

Offset	Byte Length	Value	Description
0x30	1	0x03	Code Type
0x31	1	XX	Indicator. Bit 7 clear means another image follows. Bit 7 set means that this image is the last image in the PCI Option ROM. Bits 0–6 are reserved.
		0x00 0x80	Additional images follow this image in the PCI Option ROM This image is the last image in the PCI Option ROM
0x32	2	0x0000	Reserved
0x34	X	XXXX	The beginning of the PCI Device Driver's PE/COFF Image

12.4.3 Nonvolatile Storage

A PCI adapter may contain some form of nonvolatile storage. Since there are no standard access mechanisms for nonvolatile storage on PCI adapters, the PCI I/O Protocol does not provide any services for nonvolatile storage. However, a PCI Device Driver may choose to implement its own access mechanisms. If there is a private channel between a PCI Controller and a nonvolatile storage device, a PCI Device Driver can use it for configuration options or vital product data.

NOTE

The fields `RomImage` *and* `RomSize` *in the PCI I/O Protocol do not provide direct access to the PCI Option ROM on a PCI adapter. Instead, they provide access to a copy of the PCI Option ROM in memory. If the contents of the* `RomImage` *are modified, only the memory copy is updated. If a vendor wishes to update the contents of a PCI Option ROM, they must provide their own utility or driver to perform this task. There is no guarantee that the BAR for the PCI Option ROM is valid at the time that the utility or driver may execute, so the utility or driver must provide the code required to gain write access to the PCI Option ROM contents. The algorithm for gaining write access to a PCI Option ROM is both platform specific and adapter specific, so it is outside the scope of this document.*

12.4.4 PCI Hot-Plug Events

It is possible to design a PCI Bus Driver to work with PCI Bus that conforms to the PCI Hot-Plug Specification. There are two levels of functionality that could be provided in the preboot environment. The first is to initialize the PCI Hot-Plug capable bus so it can be used by an operating system that also conforms to the PCI Hot-Plug Specification. This only affects the PCI Enumeration that is performed in either the PCI Bus Driver's initialization, or a firmware component that executes prior to the PCI Bus Driver's initialization. None of the PCI Device Drivers need to be aware of the fact that a PCI Controller may exist in a slot that is capable of a hot-plug event. Also, the addition, removal, and replacement of PCI adapters in the preboot environment would not be allowed.

The second level of functionality is to actually implement the full hot-plug capability in the PCI Bus Driver. This is not recommended because it adds a great deal of complexity to the PCI Bus Driver design with very little added value. However, there is nothing about the PCI Driver Model that would preclude this implementation. It would have to use an event based periodic timer to monitor the hot-plug capable slots, and take advantage of the **`ConnectController()`** and **`DisconnectController()`** Boot Services to dynamically start and stop the drivers that manage the PCI controller that is being added, removed, or replaced.

13
Protocols - SCSI Bus Support

The intent of this chapter is to specify a method of providing direct access to SCSI devices. This protocol provides services that allow a generic driver to produce the Block I/O protocol for SCSI disk devices, and allows an EFI utility to issue commands to any SCSI device. The main reason to provide such an access is to enable S.M.A.R.T. functionality during POST (i.e., issuing Mode Sense, Mode Select, and Log Sense to SCSI devices). This is accomplished by using a generic API such as SCSI Pass Thru. The use of this method will enable additional functionality in the future without modifying the EFI SCSI Pass Thru driver. SCSI Pass Thru is not limited to SCSI channels. It is applicable to all channel technologies that utilize SCSI commands such as SCSI, ATAPI, and Fiber Channel.

13.1 SCSI Pass Thru Protocol

This section defines the SCSI Pass Thru Protocol. This protocol allows information about a SCSI channel to be collected, and allows SCSI Request Packets to be sent to any SCSI devices on a SCSI channel even if those devices are not boot devices. This protocol is attached to the device handle of each SCSI channel in a system that the protocol supports, and can be used for diagnostics. It may also be used to build a Block I/O driver for SCSI hard drives and SCSI CD-ROM or DVD drives to allow those devices to become boot devices.

EFI_SCSI_PASS_THRU Protocol

This section provides a detailed description of the **`EFI_SCSI_PASS_THRU_PROTOCOL`**.

Summary

Provides services that allow SCSI Pass Thru commands to be sent to SCSI devices attached to a SCSI channel.

GUID

```
#define EFI_SCSI_PASS_THRU_PROTOCOL_GUID \
{ 0xa59e8fcf,0xbda0,0x43bb,0x90,0xb1,0xd3,0x73,0x2e,0xca,0xa8,0x77 }
```

Protocol Interface Structure

```
typedef struct _EFI_SCSI_PASS_THRU_PROTOCOL {
  EFI_SCSI_PASS_THRU_MODE               *Mode;
  EFI_SCSI_PASS_THRU_PASSTHRU           PassThru;
  EFI_SCSI_PASS_THRU_GET_NEXT_DEVICE    GetNextDevice;
  EFI_SCSI_PASS_THRU_BUILD_DEVICE_PATH  BuildDevicePath;
  EFI_SCSI_PASS_THRU_GET_TARGET_LUN     GetTargetLun;
  EFI_SCSI_PASS_THRU_RESET_CHANNEL      ResetChannel;
  EFI_SCSI_PASS_THRU_RESET_TARGET       ResetTarget;
} EFI_SCSI_PASS_THRU_PROTOCOL;
```

Parameters

Mode	A pointer to the **EFI_SCSI_PASS_THRU_MODE** data for this SCSI channel. Type **EFI_SCSI_PASS_THRU_MODE** is defined in "Related Definitions" below.
PassThru	Sends a SCSI Request Packet to a SCSI device that is connected to the SCSI channel. See the **PassThru()** function description.
GetNextDevice	Used to retrieve the list of legal Target IDs and LUNs for the SCSI devices on a SCSI channel. See the **GetNextDevice()** function description.
BuildDevicePath	Used to allocate and build a device path node for a SCSI device on a SCSI channel. See the **BuildDevicePath()** function description.
GetTargetLun	Used to translate a device path node to a Target ID and LUN. See the **GetTargetLun()** function description.
ResetChannel	Resets the SCSI channel. This operation resets all the SCSI devices connected to the SCSI channel. See the **ResetChannel()** function description.
ResetTarget	Resets a SCSI device that is connected to the SCSI channel. See the **ResetTarget()** function description.

The following data values in the **EFI_SCSI_PASS_THRU_MODE** interface are read-only.

ControllerName	A Null-terminated Unicode string that represents the printable name of the SCSI controller.
ChannelName	A Null-terminated Unicode string that represents the printable name of the SCSI channel.
AdapterId	The Target ID of the host adapter on the SCSI channel.
Attributes	Additional information on the attributes of the SCSI channel. See "Related Definitions" below for the list of possible attributes.

IoAlign	Supplies the alignment requirement for any buffer used in a data transfer. *IoAlign* values of 0 and 1 mean that the buffer can be placed anywhere in memory. Otherwise, *IoAlign* must be a power of 2, and the requirement is that the start address of a buffer must be evenly divisible by *IoAlign* with no remainder.

Related Definitions

```
typedef struct {
  CHAR16     *ControllerName;
  CHAR16     *ChannelName;
  UINT32     AdapterId;
  UINT32     Attributes;
  UINT32     IoAlign;
} EFI_SCSI_PASS_THRU_MODE;

#define EFI_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL    0x0001
#define EFI_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL     0x0002
#define EFI_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO  0x0004
```

EFI_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL

If this bit is set, then the **EFI_SCSI_PASS_THRU_PROTOCOL** interface is for physical devices on the SCSI channel.

EFI_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL

If this bit is set, then the **EFI_SCSI_PASS_THRU_PROTOCOL** interface is for logical devices on the SCSI channel.

EFI_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO

If this bit is set, then the **EFI_SCSI_PASS_THRU_PROTOCOL** interface supports non blocking I/O. Every **EFI_SCSI_PASS_THRU_PROTOCOL** must support blocking I/O. The support of nonblocking I/O is optional.

Description

The **EFI_SCSI_PASS_THRU_PROTOCOL** provides information about a SCSI channel and the ability to send SCI Request Packets to any SCSI device attached to that SCSI channel. The information includes the Target ID of the host controller on the SCSI channel, the attributes of the SCSI channel, the printable name for the SCSI controller, and the printable name of the SCSI channel.

The *Attributes* field of the **EFI_SCSI_PASS_THRU_PROTOCOL** interface tells if the interface is for physical SCSI devices or logical SCSI devices. Drivers for non-RAID SCSI controllers will set both the **EFI_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL**, and the **EFI_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL** bits. Drivers for RAID controllers that allow access to the physical devices and logical devices will produce two **EFI_SCSI_PASS_THRU_PROTOCOL** interfaces. One with the just the **EFI_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL** bit set and another with just the **EFI_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL** bit set. One interface can be used to access the physical devices attached to the RAID controller, and the other can be used to access the logical devices attached to the RAID controller for its current configuration. Drivers for RAID controllers that do not allow access to the physical devices will produce one **EFI_SCSI_PASS_THROUGH_PROTOCOL** interface with just the **EFI_SCSI_PASS_THRU_LOGICAL** bit set. The interface for logical devices can also be used by a file system driver to mount the RAID volumes. An **EFI_SCSI_PASS_THRU_PROTOCOL** with neither **EFI_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL** nor **EFI_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL** set is an illegal configuration.

The Attributes field also contains the **EFI_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO** bit. All **EFI_SCSI_PASS_THRU_PROTOCOL** interfaces must support blocking I/O. If this bit is set, then the interface support both blocking I/O and nonblocking I/O.

Each **EFI_SCSI_PASS_THRU_PROTOCOL** instance must have an associated device path. Typically this will have an *ACPI* device path node and a *PCI* device path node, although variations will exist. For a SCSI controller that supports only one channel per PCI bus/device/function, it is recommended, but not required, that an additional *Controller* device path node (for controller 0) be appended to the device path. For a SCSI controller that supports multiple channels per PCI bus/device/function, it is required that a *Controller* device path node be appended for each channel.

Additional information about the SCSI channel can be obtained from protocols attached to the same handle as the **EFI_SCSI_PASS_THRU_PROTOCOL**, or one of its parent handles. This would include the device I/O abstraction used to access the internal registers and functions of the SCSI controller.

EFI_SCSI_PASS_THRU_PROTOCOL.PassThru()

Summary

Sends a SCSI Request Packet to a SCSI device that is attached to the SCSI channel. This function supports both blocking I/O and nonblocking I/O. The blocking I/O functionality is required, and the nonblocking I/O functionality is optional.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_PASS_THRU_PASSTHRU) (
  IN     EFI_SCSI_PASS_THRU_PROTOCOL              *This,
  IN     UINT32                                   Target,
  IN     UINT64                                   Lun,
  IN OUT EFI_SCSI_PASS_THRU_SCSI_REQUEST_PACKET   *Packet,
  IN     EFI_EVENT                                Event    OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_SCSI_PASS_THRU_PROTOCOL** instance. Type **EFI_SCSI_PASS_THRU_PROTOCOL** is defined in Section 13.1.
Target	The Target ID of the SCSI device to send the SCSI Request Packet.
Lun	The LUN of the SCSI device to send the SCSI Request Packet.
Packet	A pointer to the SCSI Request Packet to send to the SCSI device specified by *Target* and *Lun*. See "Related Definitions" below for a description of **EFI_SCSI_PASS_THRU_SCSI_REQUEST_PACKET**.
Event	If nonblocking I/O is not supported then *Event* is ignored, and blocking I/O is performed. If *Event* is **NULL**, then blocking I/O is performed. If *Event* is not **NULL** and non blocking I/O is supported, then nonblocking I/O is performed, and *Event* will be signaled when the SCSI Request Packet completes.

Related Definitions

```
typedef struct {
  UINT64     Timeout;
  VOID       *DataBuffer;
  VOID       *SenseData;
  VOID       *Cdb;
  UINT32     TransferLength;
  UINT8      CdbLength;
  UINT8      DataDirection;
  UINT8      HostAdapterStatus;
  UINT8      TargetStatus;
  UINT8      SenseDataLength;
} EFI_SCSI_PASS_THRU_SCSI_REQUEST_PACKET;
```

Timeout	The timeout, in 100 ns units, to use for the execution of this SCSI Request Packet. A *Timeout* value of 0 means that this function will wait indefinitely for the SCSI Request Packet to execute. If *Timeout* is greater than zero, then this function will return **EFI_TIMEOUT** if the time required to execute the SCSI Request Packet is greater than *Timeout*.
DataBuffer	A pointer to the data buffer to transfer between the SCSI controller and the SCSI device. Must be aligned to the boundary specified in the *IoAlign* field of the **EFI_SCSI_PASS_THRU_MODE** structure.
SenseData	A pointer to the sense data that was generated by the execution of the SCSI Request Packet. Must be aligned to the boundary specified in the *IoAlign* field of the **EFI_SCSI_PASS_THRU_MODE** structure.
Cdb	A pointer to buffer that contains the Command Data Block to send to the SCSI device specified by *Target* and *Lun*.
TransferLength	On input, the size, in bytes, of *DataBuffer*. On output, the number of bytes transferred between the SCSI controller and the SCSI device. If *TransferLength* is larger than the SCSI controller can handle, no data will be transferred, *TransferLength* will be updated to contain the number of bytes that the SCSI controller is able to transfer, and **EFI_BAD_BUFFER_SIZE** will be returned.
CdbLength	The length, in bytes, of the buffer *Cdb*. The standard values are 6, 10, 12, and 16, but other values are possible if a variable length CDB is used.
DataDirection	The direction of the data transfer. 0 for reads, 1 for writes. All other values are reserved, and must not be used.

HostAdapterStatus	The status of the host adapter specified by *This* when the SCSI Request Packet was executed on the target device. See the possible values listed below. If bit 7 of this field is set, then *HostAdapterStatus* is a vendor defined error code.
TargetStatus	The status returned by the device specified by *Target* and *Lun* when the SCSI Request Packet was executed. See the possible values listed below.
SenseDataLength	On input, the length in bytes of the *SenseData* buffer. On output, the number of bytes written to the *SenseData* buffer.

```
//
// HostAdapterStatus
//
#define EFI_SCSI_STATUS_HOST_ADAPTER_OK                      0x00
#define EFI_SCSI_STATUS_HOST_ADAPTER_TIMEOUT_COMMAND         0x09
#define EFI_SCSI_STATUS_HOST_ADAPTER_TIMEOUT                 0x0b
#define EFI_SCSI_STATUS_HOST_ADAPTER_MESSAGE_REJECT          0x0d
#define EFI_SCSI_STATUS_HOST_ADAPTER_BUS_RESET               0x0e
#define EFI_SCSI_STATUS_HOST_ADAPTER_PARITY_ERROR            0x0f
#define EFI_SCSI_STATUS_HOST_ADAPTER_REQUEST_SENSE_FAILED    0x10
#define EFI_SCSI_STATUS_HOST_ADAPTER_SELECTION_TIMEOUT       0x11
#define EFI_SCSI_STATUS_HOST_ADAPTER_DATA_OVERRUN_UNDERRUN   0x12
#define EFI_SCSI_STATUS_HOST_ADAPTER_BUS_FREE                0x13
#define EFI_SCSI_STATUS_HOST_ADAPTER_PHASE_ERROR             0x14
#define EFI_SCSI_STATUS_HOST_ADAPTER_OTHER                   0x7f

//
// TargetStatus
//
#define EFI_SCSI_STATUS_TARGET_GOOD                          0x00
#define EFI_SCSI_STATUS_TARGET_CHECK_CONDITION               0x02
#define EFI_SCSI_STATUS_TARGET_CONDITION_MET                 0x04
#define EFI_SCSI_STATUS_TARGET_BUSY                          0x08
#define EFI_SCSI_STATUS_TARGET_INTERMEDIATE                  0x10
#define EFI_SCSI_STATUS_TARGET_INTERMEDIATE_CONDITION_MET    0x14
#define EFI_SCSI_STATUS_TARGET_RESERVATION_CONFLICT          0x18
#define EFI_SCSI_STATUS_TARGET_COMMAND_TERMINATED            0x22
#define EFI_SCSI_STATUS_TARGET_QUEUE_FULL                    0x28
```

Description

The **EFI_SCSI_PASS_THRU_PROTOCOL.PassThru()** function sends the SCSI Request Packet specified by *Packet* to the SCSI device specified by *Target* and *Lun*. If the driver supports nonblocking I/O and *Event* is not **NULL**, then the driver will return immediately after the command is sent to the selected device, and will later signal *Event* when the command has completed. If the driver supports nonblocking I/O and *Event* is **NULL**, then the driver will send the command to the selected device and block until it is complete. If the driver does not support nonblocking I/O, the *Event* parameter is ignored, and the driver will send the command to the selected device and block until it is complete.

If *Packet* is successfully sent to the SCSI device, then **EFI_SUCCESS** is returned.

If *Packet* cannot be sent because there are too many packets already queued up, then **EFI_NOT_READY** is returned. The caller may retry *Packet* at a later time.

If a device error occurs while sending the *Packet*, then **EFI_DEVICE_ERROR** is returned.

If a timeout occurs during the execution of *Packet*, then **EFI_TIMEOUT** is returned.

If *Target* or *Lun* are not in a valid range for the SCSI channel, then **EFI_INVALID_PARAMETER** is returned. If *DataBuffer* or *SenseData* do not meet the alignment requirement specified by the *IoAlign* field of the **EFI_SCSI_PASS_THRU_MODE** structure, then **EFI_INVALID_PARAMETER** is returned. If any of the other fields of *Packet* are invalid, then **EFI_INVALID_PARAMETER** is returned.

If the data buffer described by *DataBuffer* and *TransferLength* is too big to be transferred in a single command, then no data is transferred and **EFI_BAD_BUFFER_SIZE** is returned. The number of bytes that can be transferred in a single command are returned in *TransferLength*.

If the command described in *Packet* is not supported by the host adapter, then **EFI_UNSUPPORTED** is returned.

If **EFI_SUCCESS**, **EFI_WARN_BUFFER_TOO_SMALL**, **EFI_DEVICE_ERROR**, or **EFI_TIMEOUT** is returned, then the caller must examine the status fields in *Packet* in the following precedence order: *HostAdapterStatus* followed by *TargetStatus* followed by *SenseDataLength*, followed by *SenseData*. If nonblocking I/O is being used, then the status fields in *Packet* will not be valid until the *Event* associated with *Packet* is signaled.

If **EFI_NOT_READY**, **EFI_INVALID_PARAMETER** or **EFI_UNSUPPORTED** is returned, then *Packet* was never sent, so the status fields in *Packet* are not valid. If nonblocking I/O is being used, the *Event* associated with *Packet* will not be signaled.

Status Codes Returned

EFI_SUCCESS	The SCSI Request Packet was sent by the host, and *TransferLength* bytes were transferred to/from *DataBuffer*. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.
EFI_BAD_BUFFER_SIZE	The SCSI Request Packet was not executed. The number of bytes that could be transferred is returned in *TransferLength*. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.
EFI_NOT_READY	The SCSI Request Packet could not be sent because there are too many SCSI Request Packets already queued. The caller may retry again later.
EFI_DEVICE_ERROR	A device error occurred while attempting to send the SCSI Request Packet. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.
EFI_INVALID_PARAMETER	*Target*, *Lun*, or the contents of *ScsiRequestPacket* are invalid. The SCSI Request Packet was not sent, so no additional status information is available.
EFI_UNSUPPORTED	The command described by the SCSI Request Packet is not supported by the host adapter. The SCSI Request Packet was not sent, so no additional status information is available.
EFI_TIMEOUT	A timeout occurred while waiting for the SCSI Request Packet to execute. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.

EFI_SCSI_PASS_THRU_PROTOCOL.GetNextDevice()

Summary

Used to retrieve the list of legal Target IDs and LUNs for SCSI devices on a SCSI channel. These can either be the list SCSI devices that are actually present on the SCSI channel, or the list of legal Target Ids and LUNs for the SCSI channel. Regardless, the caller of this function must probe the Target ID and LUN returned to see if a SCSI device is actually present at that location on the SCSI channel.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_PASS_THRU_GET_NEXT_DEVICE) (
  IN     EFI_SCSI_PASS_THRU_PROTOCOL    *This,
  IN OUT UINT32                         *Target,
  IN OUT UINT64                         *Lun
  );
```

Parameters

This	A pointer to the **EFI_SCSI_PASS_THRU_PROTOCOL** instance. Type **EFI_SCSI_PASS_THRU_PROTOCOL** is defined in Section 13.1.
Target	On input, a pointer to the Target ID of a SCSI device present on the SCSI channel. On output, a pointer to the Target ID of the next SCSI device present on a SCSI channel. An input value of **0xFFFFFFFF** retrieves the Target ID of the first SCSI device present on a SCSI channel.
Lun	On input, a pointer to the LUN of a SCSI device present on the SCSI channel. On output, a pointer to the LUN of the next SCSI device present on a SCSI channel.

Description

The **EFI_SCSI_PASS_THRU_PROTOCOL.GetNextDevice()** function retrieves the Target ID and LUN of a SCSI device present on a SCSI channel. If *Target* is **0xFFFFFFFF**, then the Target ID and LUN of the first SCSI device is returned in *Target* and *Lun* and **EFI_SUCCESS** is returned. If *Target* and *Lun* is a Target ID and LUN value that was returned on a previous call to **GetNextDevice()**, then the Target ID and LUN of the next SCSI device on the SCSI channel is returned in *Target* and *Lun*, and **EFI_SUCCESS** is returned. If *Target* is not **0xFFFFFFFF**, and *Target* and *Lun* were not returned on a previous call to **GetNextDevice()**, then **EFI_INVALID_PARAMETER** is returned. If *Target* and *Lun* are the Target ID and LUN of the last SCSI device on the SCSI channel, then **EFI_NOT_FOUND** is returned.

Status Codes Returned

EFI_SUCCESS	The Target ID and LUN of the next SCSI device on the SCSI channel was returned in *Target* and *Lun*.
EFI_NOT_FOUND	There are no more SCSI devices on this SCSI channel.
EFI_INVALID_PARAMETER	*Target* is not **0xFFFFFFFF**, and *Target* and *Lun* were not returned on a previous call to **GetNextDevice()**.

EFI_SCSI_PASS_THRU_PROTOCOL.BuildDevicePath()

Summary

Used to allocate and build a device path node for a SCSI device on a SCSI channel.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_PASS_THRU_BUILD_DEVICE_PATH) (
  IN     EFI_SCSI_PASS_THRU_PROTOCOL    *This,
  IN     UINT32                         Target,
  IN     UINT64                         Lun
  IN OUT EFI_DEVICE_PATH_PROTOCOL       **DevicePath
  );
```

Parameters

This	A pointer to the **EFI_SCSI_PASS_THRU_PROTOCOL** instance. Type **EFI_SCSI_PASS_THRU_PROTOCOL** is defined in Section 13.1.
Target	The Target ID of the SCSI device for which a device path node is to be allocated and built.
Lun	The LUN of the SCSI device for which a device path node is to be allocated and built.
DevicePath	A pointer to a single device path node that describes the SCSI device specified by *Target* and *Lun*. This function is responsible for allocating the buffer *DevicePath* with the boot service **AllocatePool()**. It is the caller's responsibility to free *DevicePath* when the caller is finished with *DevicePath*.

Description

The **EFI_SCSI_PASS_THRU_PROTOCOL.BuildDevicePath()** function allocates and builds a single device path node for the SCSI device specified by *Target* and *Lun*. If the SCSI device specified by *Target* and *Lun* are not present on the SCSI channel, then **EFI_NOT_FOUND** is returned. If *DevicePath* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If there are not enough resources to allocate the device path node, then **EFI_OUT_OF_RESOURCES** is returned. Otherwise, *DevicePath* is allocated with the boot service **AllocatePool()**, the contents of *DevicePath* are initialized to describe the SCSI device specified by *Target* and *Lun*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The device path node that describes the SCSI device specified by *Target* and *Lun* was allocated and returned in *DevicePath*.
EFI_NOT_FOUND	The SCSI devices specified by *Target* and *Lun* does not exist on the SCSI channel.
EFI_INVALID_PARAMETER	*DevicePath* is **NULL**.
EFI_OUT_OF_RESOURCES	There are not enough resources to allocate *DevicePath*.

EFI_SCSI_PASS_THRU_PROTOCOL.GetTargetLun()

Summary

Used to translate a device path node to a Target ID and LUN.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_PASS_THRU_GET_TARGET_LUN) (
  IN   EFI_SCSI_PASS_THRU_PROTOCOL     *This,
  IN   EFI_DEVICE_PATH_PROTOCOL        *DevicePath
  OUT  UINT32                          *Target,
  OUT  UINT64                          *Lun
  );
```

Parameters

This	A pointer to the **EFI_SCSI_PASS_THRU_PROTOCOL** instance. Type **EFI_SCSI_PASS_THRU_PROTOCOL** is defined in Section 13.1.
DevicePath	A pointer to the device path node that describes a SCSI device on the SCSI channel.
Target	A pointer to the Target ID of a SCSI device on the SCSI channel.
Lun	A pointer to the LUN of a SCSI device on the SCSI channel.

Description

The **EFI_SCSI_PASS_THRU_PROTOCOL.GetTargetLun()** function determines the Target ID and LUN associated with the SCSI device described by *DevicePath*. If *DevicePath* is a device path node type that the SCSI Pass Thru driver supports, then the SCSI Pass Thru driver will attempt to translate the contents *DevicePath* into a Target ID and LUN. If this translation is successful, then that Target ID and LUN are returned in *Target* and *Lun*, and **EFI_SUCCESS** is returned.

If *DevicePath*, *Target*, or *Lun* are **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *DevicePath* is not a device path node type that the SCSI Pass Thru driver supports, then **EFI_UNSUPPORTED** is returned.

If *DevicePath* is a device path node type that the SCSI Pass Thru driver supports, but there is not a valid translation from *DevicePath* to a Target ID and LUN, then **EFI_NOT_FOUND** is returned.

Status Codes Returned

EFI_SUCCESS	*DevicePath* was successfully translated to a Target ID and LUN, and they were returned in *Target* and *Lun*.
EFI_INVALID_PARAMETER	*DevicePath* is **NULL**.
EFI_INVALID_PARAMETER	*Target* is **NULL**.
EFI_INVALID_PARAMETER	*Lun* is **NULL**.
EFI_UNSUPPORTED	This driver does not support the device path node type in *DevicePath*.
EFI_NOT_FOUND	A valid translation from *DevicePath* to a Target ID and LUN does not exist.

EFI_SCSI_PASS_THRU_PROTOCOL.ResetChannel()

Summary

Resets a SCSI channel. This operation resets all the SCSI devices connected to the SCSI channel.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_PASS_THRU_RESET_CHANNEL) (
  IN EFI_SCSI_PASS_THRU_PROTOCOL    *This
  );
```

Parameters

This	A pointer to the **EFI_SCSI_PASS_THRU_PROTOCOL** instance. Type **EFI_SCSI_PASS_THRU_PROTOCOL** is defined in Section 13.1.

Description

The **EFI_SCSI_PASS_THRU_PROTOCOL.ResetChannel()** function resets a SCSI channel. This operation resets all the SCSI devices connected to the SCSI channel. If this SCSI channel does not support a reset operation, then **EFI_UNSUPPORTED** is returned. If a device error occurs while executing that channel reset operation, then **EFI_DEVICE_ERROR** is returned. If a timeout occurs during the execution of the channel reset operation, then **EFI_TIMEOUT** is returned. If the channel reset operation is completed, then **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The SCSI channel was reset.
EFI_UNSUPPORTED	The SCSI channel does not support a channel reset operation.
EFI_DEVICE_ERROR	A device error occurred while attempting to reset the SCSI channel.
EFI_TIMEOUT	A timeout occurred while attempting to reset the SCSI channel.

EFI_SCSI_PASS_THRU_PROTOCOL.ResetTarget()

Summary

Resets a SCSI device that is connected to a SCSI channel.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_PASS_THRU_RESET_TARGET) (
  IN EFI_SCSI_PASS_THRU_PROTOCOL     *This,
  IN UINT32                          Target,
  IN UINT64                          Lun
  );
```

Parameters

This	A pointer to the **EFI_SCSI_PASS_THRU_PROTOCOL** instance. Type **EFI_SCSI_PASS_THRU_PROTOCOL** is defined in Section 13.1.
Target	The Target ID of the SCSI device to reset.
Lun	The LUN of the SCSI device to reset.

Description

The **EFI_SCSI_PASS_THRU_PROTOCOL.ResetTarget()** function resets the SCSI device specified by *Target* and *Lun*. If this SCSI channel does not support a target reset operation, then **EFI_UNSUPPORTED** is returned. If *Target* or *Lun* are not in a valid range for this SCSI channel, then **EFI_INVALID_PARAMETER** is returned. If a device error occurs while executing that target reset operation, then **EFI_DEVICE_ERROR** is returned. If a timeout occurs during the execution of the target reset operation, then **EFI_TIMEOUT** is returned. If the target reset operation is completed, then **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The SCSI device specified by *Target* and *Lun* was reset.
EFI_UNSUPPORTED	The SCSI channel does not support a target reset operation.
EFI_INVALID_PARAMETER	*Target* or *Lun* are invalid.
EFI_DEVICE_ERROR	A device error occurred while attempting to reset the SCSI device specified by *Target* and *Lun*.
EFI_TIMEOUT	A timeout occurred while attempting to reset the SCSI device specified by *Target* and *Lun*.

13.2 SCSI Pass Thru Device Paths

An **EFI_SCSI_PASS_THRU_PROTOCOL** must be installed on a handle for its services to be available to EFI Drivers and EFI Applications. In addition to the **EFI_SCSI_PASS_THRU_PROTOCOL**, an **EFI_DEVICE_PATH_PROTOCOL** must also be installed on the same handle. See Chapter 8 for a detailed description of the **EFI_DEVICE_PATH_PROTOCOL**.

A device path describes the location of a hardware component in a system from the processor's point of view. This includes the list of busses that lie between the processor and the SCSI controller. The *EFI Specification* takes advantage of the *ACPI Specification* to name system components. For the following set of examples, a PCI SCSI controller is assumed. The examples will show a SCSI controller on the root PCI bus, and a SCSI controller behind a PCI-PCI bridge. In addition, an example of a multichannel SCSI controller will be shown.

Table 13-1 shows an example device path for a single channel PCI SCSI controller that is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. This device path consists of an ACPI Device Path Node, a PCI Device Path Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(7|0).
```

Table 13-1. Single Channel PCI SCSI Controller

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x00	PCI Function
0x11	0x01	0x07	PCI Device
0x12	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x13	0x01	0xFF	Sub type – End of Entire Device Path
0x14	0x02	0x04	Length – 0x04 bytes

Table 13-2 shows an example device path for a single channel PCI SCSI controller that is located behind a PCI to PCI bridge at PCI device number 0x07 and PCI function 0x00. The PCI to PCI bridge is directly attached to a PCI root bridge, and it is at PCI device number 0x05 and PCI function 0x00. This device path consists of an ACPI Device Path Node, two PCI Device Path Nodes, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(5|0)/PCI(7|0).
```

Table 13-2. Single Channel PCI SCSI Controller behind a PCI Bridge

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x00	PCI Function
0x11	0x01	0x05	PCI Device
0x12	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x13	0x01	0x01	Sub type – PCI
0x14	0x02	0x06	Length – 0x06 bytes
0x16	0x01	0x00	PCI Function
0x17	0x01	0x07	PCI Device
0x18	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x19	0x01	0xFF	Sub type – End of Entire Device Path
0x1A	0x02	0x04	Length – 0x04 bytes

Table 13-3 shows an example device path for channel #3 of a four channel PCI SCSI controller that is located behind a PCI to PCI bridge at PCI device number 0x07 and PCI function 0x00. The PCI to PCI bridge is directly attached to a PCI root bridge, and it is at PCI device number 0x05 and PCI function 0x00. This device path consists of an ACPI Device Path Node, two PCI Device Path Nodes, a Controller Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation of the device paths for all four of the SCSI channels are listed below. Table 2-3 shows the last device path listed.

```
ACPI(PNP0A03,0)/PCI(5|0)/PCI(7|0)/Controller(0).
ACPI(PNP0A03,0)/PCI(5|0)/PCI(7|0)/Controller(1).
ACPI(PNP0A03,0)/PCI(5|0)/PCI(7|0)/Controller(2).
ACPI(PNP0A03,0)/PCI(5|0)/PCI(7|0)/Controller(3).
```

Table 13-3. Channel #3 of a PCI SCSI Controller behind a PCI Bridge

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x00	PCI Function
0x11	0x01	0x05	PCI Device
0x12	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x13	0x01	0x01	Sub type – PCI
0x14	0x02	0x06	Length – 0x06 bytes
0x16	0x01	0x00	PCI Function
0x17	0x01	0x07	PCI Device
0x18	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x19	0x01	0x05	Sub type – Controller
0x1A	0x02	0x08	Length – 0x08 bytes
0x1C	0x04	0x0003	**Controller Number**
0x20	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x21	0x01	0xFF	Sub type – End of Entire Device Path
0x22	0x02	0x04	Length – 0x04 bytes

14
Protocols - USB Support

14.1 USB Host Controller Protocol

These sections (Sections 14.1 and below) describe the USB Host Controller Protocol. This protocol provides an I/O abstraction for a USB Host Controller. A USB Host Controller is a hardware component that interfaces to a Universal Serial Bus (USB). It moves data between system memory and devices on the USB by processing data structures and generating transactions on the USB. This protocol is used by a USB Bus Driver to perform all data transaction over the Universal Serial Bus. It also provides services to manage the USB root hub that is integrated into the USB Host Controller. USB device drivers do not use this protocol directly. Instead, they use the I/O abstraction produced by the USB Bus Driver. This protocol should only be used by drivers that require direct access to the USB bus.

14.1.1 USB Host Controller Protocol Overview

The USB Host Controller Protocol is used by code, typically USB bus drivers, running in the EFI boot services environment, to perform data transactions over a USB bus. In addition, it provides an abstraction for the root hub of the USB bus.

The interfaces provided in the **EFI_USB_HC_PROTOCOL** are used to manage data transactions on a USB bus. It also provides control methods for the USB root hub. The **EFI_USB_HC_PROTOCOL** is designed to support USB 1.1–compliant host controllers.

The **EFI_USB_HC_PROTOCOL** abstracts basic functionality that is designed to operate with both the UHCI and OHCI standards. By using this protocol, a single USB bus driver can be implemented without knowing if the underlying USB host controller conforms to the OHCI or the UHCI standards.

Each instance of the **EFI_USB_HC_PROTOCOL** corresponds to a USB host controller in a platform. The protocol is attached to the device handle of a USB host controller that is created by a device driver for the USB host controller's parent bus type. For example, a USB host controller that is implemented as a PCI device would require a PCI device driver to produce an instance of the **EFI_USB_HC_PROTOCOL**.

EFI_USB_HC_PROTOCOL

Summary

Provides basic USB host controller management, basic data transactions over USB bus, and USB root hub access.

GUID

```
#define EFI_USB_HC_PROTOCOL_GUID  \
  {0xF5089266,0x1AA0,0x4953,0x97,0xD8,0x56,0x2F,0x8A,0x73,0xB5,0x19}
```

Protocol Interface Structure

```
typedef struct _EFI_USB_HC_PROTOCOL {
   EFI_USB_HC_PROTOCOL_RESET                Reset;
   EFI_USB_HC_PROTOCOL_GET_STATE            GetState;
   EFI_USB_HC_PROTOCOL_SET_STATE            SetState;
   EFI_USB_HC_PROTOCOL_CONTROL_TRANSFER     ControlTransfer;
   EFI_USB_HC_PROTOCOL_BULK_TRANSFER        BulkTransfer;
   EFI_USB_HC_PROTOCOL_ASYNC_INTERRUPT_TRANSFER
                                            AsyncInterruptTransfer;
   EFI_USB_HC_PROTOCOL_SYNC_INTERRUPT_TRANSFER
                                            SyncInterruptTransfer;
   EFI_USB_HC_PROTOCOL_ISOCHRONOUS_TRANSFER
                                            IsochronousTransfer;
   EFI_USB_HC_PROTOCOL_ASYNC_ISOCHRONOUS_TRANSFER
                                            AsyncIsochronousTransfer;
   EFI_USB_HC_PROTOCOL_GET_ROOTHUB_PORT_NUMBER
                                            GetRootHubPortNumber;
   EFI_USB_HC_PROTOCOL_GET_ROOTHUB_PORT_STATUS
                                            GetRootHubPortStatus;
   EFI_USB_HC_PROTOCOL_SET_ROOTHUB_PORT_FEATURE
                                            SetRootHubPortFeature;
   EFI_USB_HC_PROTOCOL_CLEAR_ROOTHUB_PORT_FEATURE
                                            ClearRootHubPortFeature;
   UINT16                                   MajorRevision;
   UINT16                                   MinorRevision;
} EFI_USB_HC_PROTOCOL;
```

Parameters

Reset	Software reset of USB. See the **Reset()** function description.
GetState	Retrieves the current state of the USB host controller. See the **GetState()** function description.
SetState	Sets the USB host controller to a specific state. See the **SetState()** function description.
ControlTransfer	Submits a control transfer to a target USB device. See the **ControlTransfer()** function description.

BulkTransfer	Submits a bulk transfer to a bulk endpoint of a USB device. See the **BulkTransfer()** function description.
AsyncInterruptTransfer	Submits an asynchronous interrupt transfer to an interrupt endpoint of a USB device. See the **AsyncInterruptTransfer()** function description.
SyncInterruptTransfer	Submits a synchronous interrupt transfer to an interrupt endpoint of a USB device. See the **SyncInterruptTransfer()** function description.
IsochronousTransfer	Submits isochronous transfer to an isochronous endpoint of a USB device. See the **IsochronousTransfer()** function description.
AsyncIsochronousTransfer	Submits nonblocking USB isochronous transfer. See the **AsyncIsochronousTransfer()** function description.
GetRootHubPortNumber	Retrieves the number of root hub ports that are produced by the USB host controller. See the **GetRootHubPortNumber()** function description.
GetRootHubPortStatus	Retrieves the status of the specified root hub port. See the **GetRootHubPortStatus()** function description.
SetRootHubPortFeature	Sets the feature for the specified root hub port. See the **SetRootHubPortFeature()** function description.
ClearRootHubPortFeature	Clears the feature for the specified root hub port. See the **ClearRootHubPortFeature()** function description.
MajorRevision	The major revision number of the USB host controller. The revision information indicates the release of the Universal Serial Bus Specification with which the host controller is compliant.
MinorRevision	The minor revision number of the USB host controller. The revision information indicates the release of the Universal Serial Bus Specification with which the host controller is compliant.

Description

The **EFI_USB_HC_PROTOCOL** provides USB host controller management, basic data transactions over a USB bus, and USB root hub access. A device driver that wishes to manage a USB bus in a system retrieves the **EFI_USB_HC_PROTOCOL** instance that is associated with the USB bus to be managed. A device handle for a USB host controller will minimally contain an **EFI_DEVICE_PATH_PROTOCOL** instance, and an **EFI_USB_HC_PROTOCOL** instance.

EFI_USB_HC_PROTOCOL.Reset()

Summary

Provides software reset for the USB host controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_RESET) (
  IN  EFI_USB_HC_PROTOCOL     *This,
  IN  UINT16                  Attributes
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
Attributes	A bit mask of the reset operation to perform. See “Related Definitions” below for a list of the supported bit mask values.

Related Definitions

```
#define EFI_USB_HC_RESET_GLOBAL             0x0001
#define EFI_USB_HC_RESET_HOST_CONTROLLER    0x0002
```

EFI_USB_HC_RESET_GLOBAL
If this bit is set, a global reset signal will be sent to the USB bus. This resets all of the USB bus logic, including the USB host controller hardware and all the devices attached on the USB bus.

EFI_USB_HC_RESET_HOST_CONTROLLER
If this bit is set, the USB host controller hardware will be reset. No reset signal will be sent to the USB bus.

Description

This function provides a software mechanism to reset a USB host controller. The type of reset is specified by the *Attributes* parameter. If the type of reset specified by *Attributes* is not valid, then **EFI_INVALID_PARAMETER** is returned. If the reset operation is completed, then **EFI_SUCCESS** is returned. If the type of reset specified by *Attributes* is not currently supported by the host controller hardware, **EFI_UNSUPPORTD** is returned. If a device error occurs during the reset operation, then **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	The reset operation succeeded.
EFI_INVALID_PARAMETER	*Attributes* is not valid.
EFI_UNSUPPORTED	The type of reset specified by *Attributes* is not currently supported by the host controller hardware.
EFI_DEVICE_ERROR	An error was encountered while attempting to perform the reset operation.

EFI_USB_HC_PROTOCOL.GetState()

Summary

Retrieves current state of the USB host controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_GET_STATE) (
  IN  EFI_USB_HC_PROTOCOL    *This,
  OUT EFI_USB_HC_STATE       *State
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
State	A pointer to the **EFI_USB_HC_STATE** data structure that indicates current state of the USB host controller. Type **EFI_USB_HC_STATE** is defined in "Related Definitions."

Related Definitions

```
typedef enum {
  EfiUsbHcStateHalt,
  EfiUsbHcStateOperational,
  EfiUsbHcStateSuspend,
  EfiUsbHcStateMaximum
} EFI_USB_HC_STATE;
```

EfiUsbHcStateHalt

The host controller is in halt state. No USB transactions can occur while in this state. The host controller can enter this state for three reasons:

1. After host controller hardware reset.
2. Explicitly set by software.
3. Triggered by a fatal error such as consistency check failure.

EfiUsbHcStateOperational

The host controller is in an operational state. When in this state, the host controller can execute bus traffic. This state must be explicitly set to enable the USB bus traffic.

EfiUsbHcStateSuspend

The host controller is in the suspend state. No USB transactions can occur while in this state. The host controller enters this state for the following reasons:

1. Explicitly set by software.
2. Triggered when there is no bus traffic for 3 microseconds.

Description

This function is used to retrieve the USB host controller's current state. The USB Host Controller Protocol publishes three states for USB host controller, as defined in "Related Definitions" below. If *State* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If a device error occurs while attempting to retrieve the USB host controllers current state, then **EFI_DEVICE_ERROR** is returned. Otherwise, the USB host controller's current state is returned in *State*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The state information of the host controller was returned in *State*.
EFI_INVALID_PARAMETER	State is **NULL**.
EFI_DEVICE_ERROR	An error was encountered while attempting to retrieve the host controller's current state.

EFI_USB_HC_PROTOCOL.SetState()

Summary

Sets the USB host controller to a specific state.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_SET_STATE) (
  IN EFI_USB_HC_PROTOCOL     *This,
  IN EFI_USB_HC_STATE        State
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
State	Indicates the state of the host controller that will be set. See the definition and description of the type **EFI_USB_HC_STATE** in the **GetState()** function description.

Description

This function is used to explicitly set a USB host controller's state. There are three states defined for the USB host controller. These are the halt state, the operational state and the suspend state. Figure 14-1 illustrates the possible state transitions:

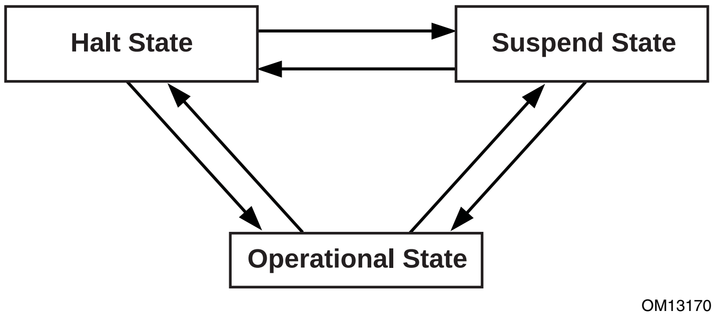


Figure 14-1. Software Triggered State Transitions of a USB Host Controller

If the state specified by *State* is not valid, then **EFI_INVALID_PARAMETER** is returned. If a device error occurs while attempting to place the USB host controller into the state specified by *State*, then **EFI_DEVICE_ERROR** is returned. If the USB host controller is successfully placed in the state specified by *State*, then **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The USB host controller was successfully placed in the state specified by *State*.
EFI_INVALID_PARAMETER	*State* is invalid.
EFI_DEVICE_ERROR	Failed to set the state specified by *State* due to device error.

EFI_USB_HC_PROTOCOL.ControlTransfer()

Summary

Submits control transfer to a target USB device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_CONTROL_TRANSFER) (
  IN      EFI_USB_HC_PROTOCOL     *This,
  IN      UINT8                   DeviceAddress,
  IN      BOOLEAN                 IsSlowDevice,
  IN      UINT8                   MaximumPacketLength,
  IN      EFI_USB_DEVICE_REQUEST   *Request,
  IN      EFI_USB_DATA_DIRECTION   TransferDirection,
  IN OUT VOID                     *Data                 OPTIONAL,
  IN OUT UINTN                    *DataLength           OPTIONAL,
  IN      UINTN                   TimeOut,
  OUT     UINT32                  *TransferResult
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
DeviceAddress	Represents the address of the target device on the USB, which is assigned during USB enumeration.
IsSlowDevice	Indicates whether the target device is slow device or full-speed device.
MaximumPacketLength	Indicates the maximum packet size that the default control transfer endpoint is capable of sending or receiving.
Request	A pointer to the USB device request that will be sent to the USB device. Refer to Section 2.5.1 of *EFI 1.1 USB Driver Model, version 0.7*.
TransferDirection	Specifies the data direction for the transfer. There are three values available, **EfiUsbDataIn**, **EfiUsbDataOut** and **EfiUsbNoData**. Refer to Section 2.5.1 of *EFI1.1 USB Driver Model, version 0.7*.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
DataLength	On input, indicates the size, in bytes, of the data buffer specified by *Data*. On output, indicates the amount of data actually transferred.

TimeOut	Indicates the maximum time, in milliseconds, which the transfer is allowed to complete.
TransferResult	A pointer to the detailed result information generated by this control transfer. Refer to Section 2.5.1 of *EFI1.1 USB Driver Model, version 0.7*.

Description

This function is used to submit a control transfer to a target USB device specified by *DeviceAddress*. Control transfers are intended to support configuration/command/status type communication flows between host and USB device.

There are three control transfer types according to the data phase. If the *TransferDirection* parameter is **EfiUsbNoData**, *Data* is **NULL**, and *DataLength* is 0, then no data phase is present in the control transfer. If the *TransferDirection* parameter is **EfiUsbDataOut**, then *Data* specifies the data to be transmitted to the device, and *DataLength* specifies the number of bytes to transfer to the device. In this case, there is an OUT DATA stage followed by a SETUP stage. If the *TransferDirection* parameter is **EfiUsbDataIn**, then *Data* specifies the data to be received from the device, and *DataLength* specifies the number of bytes to receive from the device. In this case there is an IN DATA stage followed by a SETUP stage.

If the control transfer has completed successfully, then **EFI_SUCCESS** is returned. If the transfer cannot be completed within the timeout specified by *TimeOut*, then **EFI_TIMEOUT** is returned. If an error other than timeout occurs during the USB transfer, then **EFI_DEVICE_ERROR** is returned and the detailed error code will be returned in the *TransferResult* parameter.

EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

1. *TransferDirection* is invalid.
2. *TransferDirection*, *Data*, and *DataLength* do not match one of the three control transfer types described above.
3. *Request* pointer is **NULL**.
4. *MaximumPacketLength* is not valid. If *IsSlowDevice* is **TRUE**, then *MaximumPacketLength* must be 8. If *IsSlowDevice* is **FALSE**, then *MaximumPacketLength* must be 8, 16, 32, or 64.
5. *TransferResult* pointer is **NULL**.

Status Codes Returned

EFI_SUCCESS	The control transfer was completed successfully.
EFI_OUT_OF_RESOURCES	The control transfer could not be completed due to a lack of resources.
EFI_INVALID_PARAMETER	Some parameters are invalid. The possible invalid parameters are described in “Description” above.
EFI_TIMEOUT	The control transfer failed due to timeout.
EFI_DEVICE_ERROR	The control transfer failed due to host controller or device error. Caller should check *TransferResult* for detailed error information.

EFI_USB_HC_PROTOCOL.BulkTransfer()

Summary

Submits bulk transfer to a bulk endpoint of a USB device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_BULK_TRANSFER) (
  IN      EFI_USB_HC_PROTOCOL   *This,
  IN      UINT8                 DeviceAddress,
  IN      UINT8                 EndPointAddress,
  IN      UINT8                 MaximumPacketLength,
  IN OUT  VOID                  *Data,
  IN OUT  UINTN                 *DataLength,
  IN OUT  UINT8                 *DataToggle,
  IN      UINTN                 TimeOut,
  OUT     UINT32                *TransferResult
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
DeviceAddress	Represents the address of the target device on the USB, which is assigned during USB enumeration.
EndPointAddress	The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data transfer in one direction except the control endpoint (whose default endpoint address is 0). It is the caller's responsibility to make sure that the *EndPointAddress* represents a bulk endpoint.
MaximumPacketLength	Indicates the maximum packet size the target endpoint is capable of sending or receiving.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
DataLength	When input, indicates the size, in bytes, of the data buffer specified by *Data*. When output, indicates the actually transferred data size.
DataToggle	A pointer to the data toggle value. On input, it indicates the initial data toggle value the bulk transfer should adopt; on output, it is updated to indicate the data toggle value of the subsequent bulk transfer.

TimeOut	Indicates the maximum time, in milliseconds, which the transfer is allowed to complete.
TransferResult	A pointer to the detailed result information of the bulk transfer. Refer to Section 2.5.1 of *EFI1.1 USB Driver Model, version 0.7*.

Description

This function is used to submit bulk transfer to a target endpoint of a USB device. The target endpoint is specified by *DeviceAddress* and *EndpointAddress*. Bulk transfers are designed to support devices that need to communicate relatively large amounts of data at highly variable times where the transfer can use any available bandwidth. Bulk transfers can be used only by full-speed devices.

The data transfer direction is determined by the endpoint direction that is encoded in the *EndPointAddress* parameter. Please refer to *USB Specification, Revision 1.1* on the Endpoint Address encoding.

The *DataToggle* parameter is used to track target endpoint's data sequence toggle bits. The USB provides a mechanism to guarantee data packet synchronization between data transmitter and receiver across multiple transactions. The data packet synchronization is achieved with the data sequence toggle bits and the DATA0/DATA1 PIDs. A bulk endpoint's toggle sequence is initialized to DATA0 when the endpoint experiences a configuration event. It toggles between DATA0 and DATA1 in each successive data transfer. It is host's responsibility to track the bulk endpoint's data toggle sequence and set the correct value for each data packet. The input *DataToggle* value points to the data toggle value for the first data packet of this bulk transfer; the output *DataToggle* value points to the data toggle value for the last successfully transferred data packet of this bulk transfer. The caller should record the data toggle value for use in subsequent bulk transfers to the same endpoint.

If the bulk transfer is successful, then **EFI_SUCCESS** is returned. If USB transfer cannot be completed within the timeout specified by *Timeout*, then **EFI_TIMEOUT** is returned. If an error other than timeout occurs during the USB transfer, then **EFI_DEVICE_ERROR** is returned and the detailed status code is returned in *TransferResult*.

EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

1. *Data* is **NULL**.
2. *DataLength* is 0.
3. *MaximumPacketLength* is not valid. The legal value of this parameter is 8, 16, 32, or 64.
4. *DataToggle* points to a value other than 0 and 1.
5. *TransferResult* is **NULL**.

Status Codes Returned

EFI_SUCCESS	The bulk transfer was completed successfully.
EFI_OUT_OF_RESOURCES	The bulk transfer could not be submitted due to lack of resource.
EFI_INVALID_PARAMETER	Some parameters are invalid. The possible invalid parameters are described in “Description” above.
EFI_TIMEOUT	The bulk transfer failed due to timeout.
EFI_DEVICE_ERROR	The bulk transfer failed due to host controller or device error. Caller should check *TransferResult* for detailed error information.

EFI_USB_HC_PROTOCOL.AsyncInterruptTransfer()

Summary

Submits an asynchronous interrupt transfer to an interrupt endpoint of a USB device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_ASYNC_INTERRUPT_TRANSFER) (
  IN   EFI_USB_HC_PROTOCOL                *This,
  IN   UINT8                              DeviceAddress,
  IN   UINT8                              EndPointAddress,
  IN   BOOLEAN                            IsSlowDevice,
  IN   UINT8                              MaximumPacketLength,
  IN   BOOLEAN                            IsNewTransfer,
  IN OUT UINT8                            *DataToggle,
  IN   UINTN                              PollingInterval  OPTIONAL,
  IN   UINTN                              DataLength       OPTIONAL,
  IN   EFI_ASYNC_USB_TRANSFER_CALLBACK    CallBackFunction OPTIONAL,
  IN   VOID                               *Context         OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
DeviceAddress	Represents the address of the target device on the USB, which is assigned during USB enumeration.
EndPointAddress	The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data transfer in one direction except the control endpoint (whose default endpoint address is zero). It is the caller's responsibility to make sure that the *EndPointAddress* represents an interrupt endpoint.
IsSlowDevice	Indicates whether the target device is slow device or full-speed device.
MaximumPacketLength	Indicates the maximum packet size the target endpoint is capable of sending or receiving.
IsNewTransfer	If **TRUE**, an asynchronous interrupt pipe is built between the host and the target interrupt endpoint. If **FALSE**, the specified asynchronous interrupt pipe is canceled.

DataToggle	A pointer to the data toggle value. On input, it is valid when *IsNewTransfer* is **TRUE**, and it indicates the initial data toggle value the asynchronous interrupt transfer should adopt. On output, it is valid when *IsNewTransfer* is **FALSE**, and it is updated to indicate the data toggle value of the subsequent asynchronous interrupt transfer.
PollingInterval	Indicates the interval, in milliseconds, that the asynchronous interrupt transfer is polled. This parameter is required when *IsNewTransfer* is **TRUE**.
DataLength	Indicates the length of data to be received at the rate specified by *PollingInterval* from the target asynchronous interrupt endpoint. This parameter is only required when *IsNewTransfer* is **TRUE**.
CallBackFunction	The Callback function. This function is called at the rate specified by *PollingInterval*. This parameter is only required when *IsNewTransfer* is **TRUE**. Refer to Section 2.5.3 of *EFI1.1 USB Driver Model, version 0.7*, for the definition of this type.
Context	The context that is passed to the *CallBackFunction*. This is an optional parameter and may be **NULL**.

Description

This function is used to submit asynchronous interrupt transfer to a target endpoint of a USB device. The target endpoint is specified by *DeviceAddress* and *EndpointAddress*. In the USB Specification, Revision 1.1, interrupt transfer is one of the four USB transfer types. In the **EFI_USB_HC_PROTOCOL**, interrupt transfer is divided further into synchronous interrupt transfer and asynchronous interrupt transfer.

An asynchronous interrupt transfer is typically used to query a device's status at a fixed rate. For example, keyboard, mouse, and hub devices use this type of transfer to query their interrupt endpoints at a fixed rate. The asynchronous interrupt transfer is intended to support the interrupt transfer type of "submit once, execute periodically." Unless an explicit request is made, the asychronous transfer will never retire.

If *IsNewTransfer* is **TRUE**, then an interrupt transfer is started at a fixed rate. The rate is specified by *PollingInterval*, the size of the receive buffer is specified by *DataLength*, and the callback function is specified by *CallBackFunction*. *Context* specifies an optional context that is passed to the *CallBackFunction* each time it is called. The *CallBackFunction* is intended to provide a means for the host to periodically process interrupt transfer data.

If *IsNewTransfer* is **FALSE**, then the interrupt transfer is canceled.

EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

1. Data transfer direction indicated by *EndPointAddress* is other than **EfiUsbDataIn**.
2. *IsNewTransfer* is **TRUE** and *DataLength* is 0.
3. *IsNewTransfer* is **TRUE** and *DataToggle* points to a value other than 0 and 1.
4. *IsNewTransfer* is **TRUE** and *PollingInterval* is not in the range 1..255.

Status Codes Returned

EFI_SUCCESS	The asynchronous interrupt transfer request has been successfully submitted or canceled.
EFI_INVALID_PARAMETER	Some parameters are invalid. The possible invalid parameters are described in "Description" above.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_USB_HC_PROTOCOL.SyncInterruptTransfer()

Summary

Submits synchronous interrupt transfer to an interrupt endpoint of a USB device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_SYNC_INTERRUPT_TRANSFER) (
  IN      EFI_USB_HC_PROTOCOL   *This,
  IN      UINT8                 DeviceAddress,
  IN      UINT8                 EndPointAddress,
  IN      BOOLEAN               IsSlowDevice,
  IN      UINT8                 MaximumPacketLength,
  IN OUT  VOID                  *Data,
  IN OUT  UINTN                 *DataLength,
  IN OUT  UINT8                 *DataToggle,
  IN      UINTN                 TimeOut,
  OUT     UINT32                *TransferResult
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
DeviceAddress	Represents the address of the target device on the USB, which is assigned during USB enumeration.
EndPointAddress	The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data transfer in one direction except the control endpoint (whose default endpoint address is zero). It is the caller's responsibility to make sure that the *EndPointAddress* represents an interrupt endpoint.
IsSlowDevice	Indicates whether the target device is slow device or full-speed device.
MaximumPacketLength	Indicates the maximum packet size the target endpoint is capable of sending or receiving.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
DataLength	On input, the size, in bytes, of the data buffer specified by *Data*. On output, the number of bytes transferred.

DataToggle	A pointer to the data toggle value. On input, it indicates the initial data toggle value the synchronous interrupt transfer should adopt; on output, it is updated to indicate the data toggle value of the subsequent synchronous interrupt transfer.
TimeOut	Indicates the maximum time, in milliseconds, which the transfer is allowed to complete.
TransferResult	A pointer to the detailed result information from the synchronous interrupt transfer. Refer to Section 2.5.1 of *EFI1.1 USB Driver Model, version 0.7.*

Description

This function is used to submit a synchronous interrupt transfer to a target endpoint of a USB device. The target endpoint is specified by *DeviceAddress* and *EndpointAddress*. In the USB Specification, Revision 1.1, interrupt transfer is one of the four USB transfer types. In the **EFI_USB_HC_PROTOCOL**, interrupt transfer is divided further into synchronous interrupt transfer and asynchronous interrupt transfer.

The synchronous interrupt transfer is designed to retrieve small amounts of data from a USB device through an interrupt endpoint. A synchronous interrupt transfer is only executed once for each request. This is the most significant difference from the asynchronous interrupt transfer.

If the synchronous interrupt transfer is successful, then **EFI_SUCCESS** is returned. If the USB transfer cannot be completed within the timeout specified by *Timeout*, then **EFI_TIMEOUT** is returned. If an error other than timeout occurs during the USB transfer, then **EFI_DEVICE_ERROR** is returned and the detailed status code is returned in *TransferResult*.

EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

1. Data transfer direction indicated by *EndPointAddress* is not **EfiUsbDataIn**.
2. *Data* is **NULL**.
3. *DataLength* is 0.
4. *MaximumPacketLength* is not valid. The legal value of this parameter is for the full-speed device, it should be 8, 16, 32, or 64; for the slow device, it is limited to 8.
5. *DataToggle* points to a value other than 0 and 1.
6. *TransferResult* is **NULL**.

Status Codes Returned

EFI_SUCCESS	The synchronous interrupt transfer was completed successfully.
EFI_OUT_OF_RESOURCES	The synchronous interrupt transfer could not be submitted due to lack of resource.
EFI_INVALID_PARAMETER	Some parameters are invalid. The possible invalid parameters are described in “Description” above.
EFI_TIMEOUT	The synchronous interrupt transfer failed due to timeout.
EFI_DEVICE_ERROR	The synchronous interrupt transfer failed due to host controller or device error. Caller should check *TransferResult* for detailed error information.

EFI_USB_HC_PROTOCOL.IsochronousTransfer()

Summary

Submits isochronous transfer to an isochronous endpoint of a USB device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_ISOCHRONOUS_TRANSFER) (
  IN      EFI_USB_HC_PROTOCOL   *This,
  IN      UINT8                 DeviceAddress,
  IN      UINT8                 EndPointAddress,
  IN      UINT8                 MaximumPacketLength,
  IN OUT  VOID                  *Data,
  IN      UINTN                 DataLength,
  OUT     UINT32                *TransferResult
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
DeviceAddress	Represents the address of the target device on the USB, which is assigned during USB enumeration.
EndPointAddress	The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data transfer in one direction except the control endpoint (whose default endpoint address is 0). It is the caller's responsibility to make sure that the *EndPointAddress* represents an isochronous endpoint.
MaximumPacketLength	Indicates the maximum packet size the target endpoint is capable of sending or receiving. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
DataLength	Specifies the length, in bytes, of the data to be sent to or received from the USB device.
TransferResult	A pointer to the detail result information of the isochronous transfer. Refer to Section 2.5.1 of *EFI1.1 USB Driver Model, version 0.7.*

Description

This function is used to submit isochronous transfer to a target endpoint of a USB device. The target endpoint is specified by *DeviceAddress* and *EndpointAddress*. Isochronous transfers are used when working with isochronous date. It provides periodic, continuous communication between the host and a device.

If the isochronous transfer is successful, then **EFI_SUCCESS** is returned. The isochronous transfer is designed to be completed within one USB frame time, if it cannot be completed, **EFI_TIMEOUT** is returned. If an error other than timeout occurs during the USB transfer, then **EFI_DEVICE_ERROR** is returned and the detailed status code will be returned in *TransferResult*.

EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

1. *Data* is **NULL.**
2. *DataLength* is 0.
3. *MaximumPacketLength* is larger than 1023.
4. *TransferResult* is **NULL**.

Status Codes Returned

EFI_SUCCESS	The isochronous transfer was completed successfully.
EFI_OUT_OF_RESOURCES	The isochronous transfer could not be submitted due to lack of resource.
EFI_INVALID_PARAMETER	Some parameters are invalid. The possible invalid parameters are described in "Description" above.
EFI_TIMEOUT	The isochronous transfer cannot be completed within the one USB frame time.
EFI_DEVICE_ERROR	The isochronous transfer failed due to host controller or device error. Caller should check *TransferResult* for detailed error information.

EFI_USB_HC_PROTOCOL.AsyncIsochronousTransfer()

Summary

Submits nonblocking isochronous transfer to an isochronous endpoint of a USB device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_USB_HC_PROTOCOL_ASYNC_ISOCHRONOUS_TRANSFER) (
  IN     EFI_USB_HC_PROTOCOL             *This,
  IN     UINT8                           DeviceAddress,
  IN     UINT8                           EndPointAddress,
  IN     UINT8                           MaximumPacketLength,
  IN OUT VOID                            *Data,
  IN     UINTN                           DataLength,
  IN EFI_ASYNC_USB_TRANSFER_CALLBACK     IsochronousCallBack,
  IN VOID                                *Context    OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
DeviceAddress	Represents the address of the target device on the USB, which is assigned during USB enumeration.
EndPointAddress	The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data transfer in one direction except the control endpoint (whose default endpoint address is zero). It is the caller's responsibility to make sure that the *EndPointAddress* represents an isochronous endpoint.
MaximumPacketLength	Indicates the maximum packet size the target endpoint is capable of sending or receiving. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
DataLength	Specifies the length, in bytes, of the data to be sent to or received from the USB device.

IsochronousCallback	The Callback function. This function is called if the requested isochronous transfer is completed. Refer to Section 2.5.3 of *EFI1.1 USB Driver Model, version 0.7.*
Context	Data passed to the *IsochronousCallback* function. This is an optional parameter and may be **NULL**.

Description

This is an asynchronous type of USB isochronous transfer. If the caller submits a USB isochronous transfer request through this function, this function will return immediately. When the isochronous transfer completes, the **IsochronousCallback** function will be triggered, the caller can know the transfer results. If the transfer is successful, the caller can get the data received or sent in this callback function.

The target endpoint is specified by *DeviceAddress* and *EndpointAddress*. Isochronous transfers are used when working with isochronous date. It provides periodic, continuous communication between the host and a device.

EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

1. *Data* is **NULL**.
2. *DataLength* is 0.
3. *MaximumPacketLength* is larger than 1023.

Status Codes Returned

EFI_SUCCESS	The asynchronous isochronous transfer was completed successfully.
EFI_OUT_OF_RESOURCES	The asynchronous isochronous transfer could not be submitted due to lack of resource.
EFI_INVALID_PARAMETER	Some parameters are invalid. The possible invalid parameters are described in “Description” above.

EFI_USB_HC_PROTOCOL.GetRootHubPortNumber()

Summary

Retrieves the number of root hub ports.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_GET_ROOTHUB_PORT_NUMBER) (
  IN  EFI_USB_HC_PROTOCOL   *This,
  OUT UINT8                 *PortNumber
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
PortNumber	A pointer to the number of the root hub ports.

Description

This function is used to retrieve the number of root hub ports. The number of root hub ports is required by the USB bus driver to perform bus enumeration.

Status Codes Returned

EFI_SUCCESS	The port number was retrieved successfully.
EFI_INVALID_PARAMETER	*PortNumber* is **NULL**.
EFI_DEVICE_ERROR	An error was encountered while attempting to retrieve the port number.

EFI_USB_HC_PROTOCOL.GetRootHubPortStatus()

Summary

Retrieves the current status of a USB root hub port.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_GET_ROOTHUB_PORT_STATUS) (
  IN  EFI_USB_HC_PROTOCOL        *This,
  IN  UINT8                      PortNumber,
  OUT EFI_USB_PORT_STATUS        *PortStatus
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
PortNumber	Specifies the root hub port from which the status is to be retrieved. This value is zero based. For example, if a root hub has two ports, then the first port is numbered 0, and the second port is numbered 1.
PortStatus	A pointer to the current port status bits and port status change bits. The type **EFI_USB_PORT_STATUS** is defined in "Related Definitions" below.

Related Definitions

```
typedef struct{
  UINT16   PortStatus;
  UINT16   PortChangeStatus;
} EFI_USB_PORT_STATUS;

//*****************************************************
// EFI_USB_PORT_STATUS.PortStatus bit definition
//*****************************************************
#define USB_PORT_STAT_CONNECTION      0x0001
#define USB_PORT_STAT_ENABLE          0x0002
#define USB_PORT_STAT_SUSPEND         0x0004
#define USB_PORT_STAT_OVERCURRENT     0x0008
#define USB_PORT_STAT_RESET           0x0010
#define USB_PORT_STAT_POWER           0x0100
#define USB_PORT_STAT_LOW_SPEED       0x0200
```

```
//**************************************************
// EFI_USB_PORT_STATUS.PortChangeStatus bit definition
//**************************************************
#define USB_PORT_STAT_C_CONNECTION    0x0001
#define USB_PORT_STAT_C_ENABLE        0x0002
#define USB_PORT_STAT_C_SUSPEND       0x0004
#define USB_PORT_STAT_C_OVERCURRENT   0x0008
#define USB_PORT_STAT_C_RESET         0x0010
```

PortStatus	Contains current port status bitmap. The root hub port status bitmap is unified with the USB hub port status bitmap. See Table 14-1 for a reference, which is borrowed from *Chapter 11, Hub Specification, of USB Specification, Revision 1.1.*
PortChangeStatus	Contains current port status change bitmap. The root hub port change status bitmap is unified with the USB hub port status bitmap. See Table 14-2 for a reference, which is borrowed from *Chapter 11, Hub Specification, of USB Specification, Revision 1.1.*

Table 14-1. USB Hub Port Status Bitmap

Bit	Description
0	**Current Connect Status:** (USB_PORT_STAT_CONNECTION) This field reflects whether or not a device is currently connected to this port. 0 = No device is present 1 = A device is present on this port
1	**Port Enable / Disabled:** (USB_PORT_STAT_ENABLE) Ports can be enabled by software only. Ports can be disabled by either a fault condition (disconnect event or other fault condition) or by software. 0 = Port is disabled 1 = Port is enabled
2	**Suspend:** (USB_PORT_STAT_SUSPEND) This field indicates whether or not the device on this port is suspended. 0 = Not suspended 1 = Suspended
3	**Over-current Indicator:** (USB_PORT_STAT_OVERCURRENT) This field is used to indicate that the current drain on the port exceeds the specified maximum. 0 = All no over-current condition exists on this port 1 = An over-current condition exists on this port
4	**Reset:** (USB_PORT_STAT_RESET) Indicates whether port is in reset state. 0 = Port is not in reset state 1 = Port is in reset state

continued

Table 14-1. USB Hub Port Status Bitmap (continued)

Bit	Description
5-7	**Reserved** These bits return 0 when read.
8	**Port Power:** (USB_PORT_STAT_POWER) This field reflects a port's logical, power control state. 0 = This port is in the Powered-off state 1 = This port is not in the Powered-off state
9	**Low Speed Device Attached:** (USB_PORT_STAT_LOW_SPEED) This is relevant only if a device is attached. 0 = Full-speed device attached to this port 1 = Low-speed device attached to this port
10-15	**Reserved** These bits return 0 when read.

Table 14-2. Hub Port Change Status Bitmap

Bit	Description
0	**Connect Status Change:** (USB_PORT_STAT_C_CONNECTION) Indicates a change has occurred in the port's Current Connect Status. 0 = No change has occurred to Current Connect status 1 = Current Connect status has changed
1	**Port Enable /Disable Change:** (USB_PORT_STAT_C _ENABLE) 0 = No change 1 = Port enabled/disabled status has changed
2	**Suspend Change:** (USB_PORT_STAT_C _SUSPEND) This field indicates a change in the host-visible suspend state of the attached device. 0 = No change 1 = Resume complete
3	**Over-Current Indicator Change:** (USB_PORT_STAT_C_OVERCURRENT) 0 = No change has occurred to Over-Current Indicator 1 = Over-Current Indicator has changed
4	**Reset Change:** (USB_PORT_STAT_C_RESET) This field is set when reset processing on this port is complete. 0 = No change 1 = Reset complete
5-15	**Reserved.** These bits return 0 when read.

Description

This function is used to retrieve the status of the root hub port specified by *PortNumber*.

EFI_USB_PORT_STATUS describes the port status of a specified USB port. This data structure is designed to be common to both a USB root hub port and a USB hub port.

The number of root hub ports attached to the USB host controller can be determined with the function **GetRootHubPortNumber()**. If *PortNumber* is greater than or equal to the number of ports returned by **GetRootHubPortNumber()**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, the status of the USB root hub port is returned in *PortStatus*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The status of the USB root hub port specified by *PortNumber* was returned in *PortStatus*.
EFI_INVALID_PARAMETER	*PortNumber* is invalid.

EFI_USB_HC_PROTOCOL.SetRootHubPortFeature()

Summary

Sets a feature for the specified root hub port.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_SET_ROOTHUB_PORT_FEATURE) (
  IN EFI_USB_HC_PROTOCOL                   *This,
  IN UINT8                                 PortNumber,
  IN EFI_USB_PORT_FEATURE                  PortFeature
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
PortNumber	Specifies the root hub port whose feature is requested to be set. This value is zero based. For example, if a root hub has two ports, then the first port is number 0, and the second port is numbered 1.
PortFeature	Indicates the feature selector associated with the feature set request. The port feature indicator is defined in "Related Definitions" and Table 14-3 below.

Related Definitions

```
typedef enum {
  EfiUsbPortEnable                 = 1,
  EfiUsbPortSuspend                = 2,
  EfiUsbPortReset                  = 4,
  EfiUsbPortPower                  = 8,
  EfiUsbPortConnectChange          = 16,
  EfiUsbPortEnableChange           = 17,
  EfiUsbPortSuspendChange          = 18,
  EfiUsbPortOverCurrentChange      = 19,
  EfiUsbPortResetChange            = 20
} EFI_USB_PORT_FEATURE;
```

The feature values specified in the enumeration variable have special meaning. Each value indicates its bit index in the port status and status change bitmaps, if combines these two bitmaps into a 32-bit bitmap. The meaning of each port feature is listed in Table 14-3.

Table 14-3. USB Port Feature

Port Feature	For SetRootHubPortFeature	For ClearRootHubPortFeature
EfiUsbPortEnable	Enable the given port of the root hub.	Disable the given port of the root hub.
EfiUsbPortSuspend	Put the given port into suspend state.	Restore the given port from the previous suspend state.
EfiUsbPortReset	Reset the given port of the root hub.	Clear the RESET signal for the given port of the root hub.
EfiUsbPortPower	Power the given port.	Shutdown the power from the given port.
EfiUsbPortConnectChange	N/A.	Clear USB_PORT_STAT_C_CONNECTION bit of the given port of the root hub.
EfiUsbPortEnableChange	N/A.	Clear USB_PORT_STAT_C_ENABLE bit of the given port of the root hub.
EfiUsbPortSuspendChange	N/A.	Clear USB_PORT_STAT_C_SUSPEND bit of the given port of the root hub.
EfiUsbPortOverCurrentChange	N/A.	Clear USB_PORT_STAT_C_OVERCURRENT bit of the given port of the root hub.
EfiUsbPortResetChange	N/A.	Clear USB_PORT_STAT_C_RESET bit of the given port of the root hub.

Description

This function sets the feature specified by *PortFeature* for the USB root hub port specified by *PortNumber*. Setting a feature enables that feature or starts a process associated with that feature. For the meanings about the defined features, please refer to Table 14-1 and Table 14-2.

The number of root hub ports attached to the USB host controller can be determined with the function **GetRootHubPortNumber()**. If *PortNumber* is greater than or equal to the number of ports returned by **GetRootHubPortNumber()**, then **EFI_INVALID_PARAMETER** is returned. If *PortFeature* is not **EfiUsbPortEnable**, **EfiUsbPortSuspend**, **EfiUsbPortReset** nor **EfiUsbPortPower**, then **EFI_INVALID_PARAMETER** is returned.

Status Codes Returned

EFI_SUCCESS	The feature specified by *PortFeature* was set for the USB root hub port specified by *PortNumber*.
EFI_INVALID_PARAMETER	*PortNumber* is invalid or *PortFeature* is invalid for this function.

EFI_USB_HC_PROTOCOL.ClearRootHubPortFeature()

Summary

Clears a feature for the specified root hub port.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_HC_PROTOCOL_CLEAR_ROOTHUB_PORT_FEATURE) (
  IN EFI_USB_HC_PROTOCOL    *This
  IN UINT8                  PortNumber,
  IN EFI_USB_PORT_FEATURE  PortFeature
  );
```

Parameters

This	A pointer to the **EFI_USB_HC_PROTOCOL** instance. Type **EFI_USB_HC_PROTOCOL** is defined in Section 14.1.
PortNumber	Specifies the root hub port whose feature is requested to be cleared. This value is zero-based. For example, if a root hub has two ports, then the first port is number 0, and the second port is numbered 1.
PortFeature	Indicates the feature selector associated with the feature clear request. The port feature indicator (**EFI_USB_PORT_FEATURE**) is defined in the "Related Definitions" section of the **SetRootHubPortFeature()** function description and in Table 14-3.

Description

This function clears the feature specified by *PortFeature* for the USB root hub port specified by *PortNumber*. Clearing a feature disables that feature or stops a process associated with that feature. For the meanings about the defined features, refer to Table 14-1 and Table 14-2.

The number of root hub ports attached to the USB host controller can be determined with the function **GetRootHubPortNumber()**. If *PortNumber* is greater than or equal to the number of ports returned by **GetRootHubPortNumber()**, then **EFI_INVALID_PARAMETER** is returned. If *PortFeature* is not **EfiUsbPortEnable**, **EfiUsbPortSuspend**, **EfiUsbPortPower**, **EfiUsbPortConnectChange**, **EfiUsbPortResetChange**, **EfiUsbPortEnableChange**, **EfiUsbPortSuspendChange**, or **EfiUsbPortOverCurrentChange**, then **EFI_INVALID_PARAMETER** is returned.

Status Codes Returned

EFI_SUCCESS	The feature specified by *PortFeature* was cleared for the USB root hub port specified by *PortNumber*.
EFI_INVALID_PARAMETER	*PortNumber* is invalid or *PortFeature* is invalid.

14.2 USB Driver Model

14.2.1 Scope

These sections (Sections 14.2 and below) describe the USB Driver Model. This includes the behavior of USB Bus Drivers, the behavior of a USB Device Drivers, and a detailed description of the EFI USB I/O Protocol. This document provides enough material to implement a USB Bus Driver, and the tools required to design and implement USB Device Drivers. It does not provide any information on specific USB devices.

The material contained in this document is designed to extend the *EFI Specification* and the *EFI Driver Model Specification* in a way that supports USB device drivers and USB bus drivers. These extensions are provided in the form of USB specific protocols. This document provides the information required to implement a USB Bus Driver in system firmware. The document also contains the information required by driver writers to design and implement USB Device Drivers that a platform may need to boot an EFI compliant OS.

A full understanding of the *EFI Specification*, the *EFI Driver Model Specification*, and the *USB Specification* is assumed throughout this document. The *USB* Driver Model described here is intended to be a foundation on which a *USB* Bus Driver and a wide variety of *USB* Device Drivers can be created. The current version of *USB Driver Model* is designed for USB 1.1; it may not be compatible newer revisions of USB.

14.2.2 USB Driver Model Overview

The EFI USB Driver Stack includes the USB Bus Driver, USB Host Controller Driver, and individual USB device drivers.

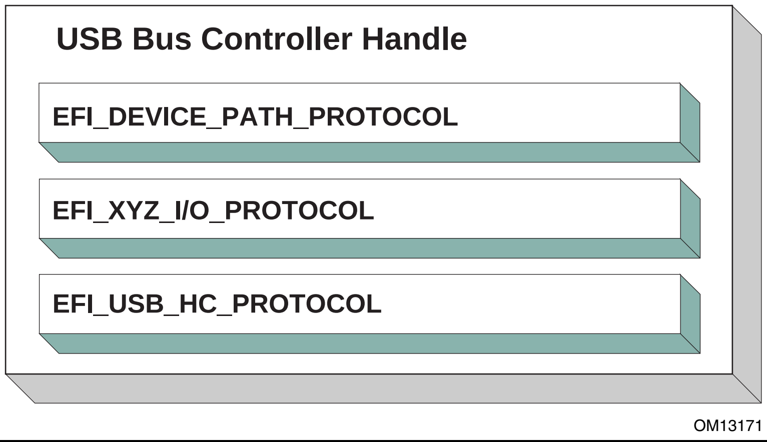


Figure 14-2. USB Bus Controller Handle

In the USB Bus Driver Design, the USB Bus Controller is managed by two drivers. One is USB Host Controller Driver, which consumes its parent bus **EFI_XYZ_IO_PROTOCOL**, and produces **EFI_USB_HC_PROTOCOL** and attaches it to the Bus Controller Handle. The other one is USB Bus Driver, which consumes **EFI_USB_HC_PROTOCOL**, and performs bus enumeration. Figure 14-2 shows protocols that are attached to the USB Bus Controller Handle. Detailed descriptions are presented in the following sections.

14.2.3 USB Bus Driver

USB Bus Driver performs periodic Enumeration on the USB Bus. In USB bus enumeration, when a new USB controller is found, the bus driver does some standard configuration for that new controller, and creates a device handle for it. The **EFI_USB_IO_PROTOCOL** and the **EFI_DEVICE_PATH** are attached to the device handle so that the USB controller can be accessed. The USB Bus Driver is also responsible for connecting USB device drivers to USB controllers. When a USB device is detached from a USB bus, the USB bus driver will stop that USB controller, and uninstall the **EFI_USB_IO_PROTOCOL** and the **EFI_DEVICE_PATH** from that handle. A detailed description is given in Section 14.2.3.3.

14.2.3.1 USB Bus Driver Entry Point

Like all other device drivers, the entry point for a USB Bus Driver attaches the **EFI_DRIVER_BINDING_PROTOCOL** to image handle of the USB Bus Driver.

14.2.3.2 Driver Binding Protocol for USB Bus Drivers

The Driver Binding Protocol contains three services. These are **Supported()**, **Start()**, and **Stop()**. **Supported()** tests to see if the USB Bus Driver can manage a device handle. A USB Bus Driver can only manage a device handle that contains **EFI_USB_HC_PROTOCOL**.

The general idea is that the USB Bus Driver is a generic driver. Since there are several types of USB Host Controllers, an **EFI_USB_HC_PROTOCOL** is used to abstract the host controller interface. Actually, a USB Bus Driver only requires an **EFI_USB_HC_PROTOCOL**.

The **Start()** function tells the USB Bus Driver to start managing the USB Bus. In this function, the USB Bus Driver creates a device handle for the root hub, and creates a timer to monitor root hub connection changes.

The **Stop()** function tells the USB Bus Driver to stop managing a USB Host Bus Controller. The **Stop()** function simply deconfigures the devices attached to the root hub. The deconfiguration is a recursive process. If the device to be deconfigured is a USB hub, then all USB devices attached to its downstream ports will be deconfigured first, then itself. If all of the child devices handles have been destroyed then the **EFI_USB_HC_PROTOCOL** is closed. Finally, the **Stop()** unction will then place the USB Host Bus Controller in a quiescent state.

14.2.3.3 USB Hot-Plug Event

Hot-Plug is one of the most important features provided by USB. A USB bus driver implements this feature through two methods. There are two types of hubs defined in the USB specification. One is the USB root hub, which is implemented in the USB Host controller. A timer event is created for the root hub. The other one is a USB Hub. An event is created for each hub that is correctly configured. All these events are associated with the same trigger which is USB bus numerator.

When USB bus enumeration is triggered, the USB Bus Driver checks the source of the event. This is required because the root hub differs from standard USB hub in checking the hub status. The status of a root hub is retrieved through the **EFI_USB_HC_PROTOCOL**, and that status of a standard USB hub is retrieved through a USB control transfer. A detailed description of the enumeration process is presented in the next section.

14.2.3.4 USB Bus Enumeration

When the periodic timer or the hubs notify event is signaled, the USB Bus Driver will perform bus numeration.

1. Determine if the event is from the root hub or a standard USB hub.
2. Determine the port on which the connection change event occurred.
3. Determine if it is a connection change or a disconnection change.
4. If a connect change is detected, then a new device has been attached. Perform the following:
   a. Reset and enable that port.
   b. Configure the new device.
   c. Parse the device configuration descriptors; get all of its interface descriptors (i.e. all USB controllers), and configure each interface.
   d. Create a new handle for each interface (USB Controller) within the USB device. Attach the **EFI_DEVICE_PATH**, and the **EFI_USB_IO_PROTOCOL** to each handle.
   e. Connect the USB Controller to a USB device driver with the Boot Service **ConnectController()** if applicable.
   f. If the USB Controller is a USB hub, create a Hub notify event which is associated with the USB Bus Enumerator, and submit an Asynchronous Interrupt Transfer Request (See Section 14.2.5).
5. If a disconnect change, then a device has been detached from the USB Bus. Perform the following:
   a. If the device is not a USB Hub, then find and deconfigure the USB Controllers within the device. Then, stop each USB controller with **DisconnectController()**, and uninstall the **EFI_DEVICE_PATH** and the **EFI_USB_IO_PROTOCOL** from the controller's handle.
   b. If the USB controller is USB hub controller, first find and deconfigure all its downstream USB devices (this is a recursive process, since there may be additional USB hub controllers on the downstream ports), then deconfigure USB hub controller itself.

14.2.4 USB Device Driver

A USB Device Driver manages a USB Controller and produces a device abstraction for use by a preboot application.

14.2.4.1 USB Device Driver Entry Point

Like all other device drivers, the entry point for a USB Device Driver attaches **`EFI_DRIVER_BINDING_PROTOCOL`** to image handle of the USB Device Driver.

14.2.4.2 Driver Binding Protocol for USB Device Drivers

The Driver Binding Protocol contains three services. These are **`Supported()`**, **`Start()`**, and **`Stop()`**.

The **`Supported()`** tests to see if the USB Device Driver can manage a device handle. This function checks to see if a controller can be managed by the USB Device Driver. This is done by opening the **`EFI_USB_IO_PROTOCOL`** bus abstraction on the USB Controller handle, and using the **`EFI_USB_IO_PROTOCOL`** services to determine if this USB Controller matches the profile that the USB Device Driver is capable of managing.

The **`Start()`** function tells the USB Device Driver to start managing a USB Controller. It opens the **`EFI_USB_IO_PROTOCOL`** instance from the handle for the USB Controller. This protocol instance is used to perform USB packet transmission over the USB bus. For example, if the USB controller is USB keyboard, then the USB keyboard driver would produce and install the **`SIMPLE_INPUT`** to the USB controller handle.

The **`Stop()`** function tells the USB Device Driver to stop managing a USB Controller. It removes the I/O abstraction protocol instance previously installed in **`Start()`** from the USB controller handle. It then closes the **`EFI_USB_IO_PROTOCOL`**.

14.2.5 EFI USB I/O Protocol Overview

This section provides a detailed description of the **`EFI_USB_IO_PROTOCOL`**. This protocol is used by code, typically drivers, running in the EFI boot services environment to access USB devices like USB keyboards, mice and mass storage devices. In particular, functions for managing devices on USB buses are defined here.

The interfaces provided in the **`EFI_USB_IO_PROTOCOL`** are for performing basic operations to access USB devices. Typically, USB devices are accessed through the four different transfers types:

• ***Controller Transfer:***	Typically used to configure the USB device into an operation mode.
• ***Interrupt Transfer:***	Typically used to get periodic small amount of data, like USB keyboard and mouse.
• ***Bulk Transfer:***	Typically used to transfer large amounts of data like reading blocks from USB mass storage devices.
• ***Isochronous Transfer:***	Typically used to transfer data at a fixed rate like voice data.

This protocol also provides mechanisms to manage and configure USB devices and controllers.

EFI_USB_IO Protocol

Summary

Provides services to manage and communicate with USB devices.

GUID

```
#define EFI_USB_IO_PROTOCOL_GUID \
  {0x2B2F68D6,0x0CD2,0x44cf,0x8E,0x8B,0xBB,0xA2,0x0B,0x1B,0x5B,0x75}
```

Protocol Interface Structure

```
typedef struct _EFI_USB_IO_PROTOCOL {
  EFI_USB_IO_CONTROL_TRANSFER           UsbControlTransfer;
  EFI_USB_IO_BULK_TRANSFER              UsbBulkTransfer;
  EFI_USB_IO_ASYNC_INTERRUPT_TRANSFER
                                  UsbAsyncInterruptTransfer;
  EFI_USB_IO_SYNC_INTERRPUT_TRANSFER    UsbSyncInterruptTransfer
  EFI_USB_IO_ISOCHRONOUS_TRANSFER       UsbIsochronousTransfer;
  EFI_USB_IO_ASYNC_ISOCHRONOUS_TRANSFER
                                  UsbAsyncIsochronousTransfer;
  EFI_USB_IO_GET_DEVICE_DESCRIPTOR      UsbGetDeviceDescriptor;
  EFI_USB_IO_GET_CONFIG_DESCRIPTOR      UsbGetConfigDescriptor;
  EFI_USB_IO_GET_INTERFACE_DESCRIPTOR
                                  UsbGetInterfaceDescriptor;
  EFI_USB_IO_GET_ENDPOINT_DESCRIPTOR    UsbGetEndpointDescriptor;
  EFI_USB_IO_GET_STRING_DESCRIPTOR      UsbGetStringDescriptor;
  EFI_USB_IO_GET_SUPPORTED_LANGUAGES    UsbGetSupportedLanguages;
  EFI_USB_IO_PORT_RESET                 UsbPortReset;
} EFI_USB_IO_PROTOCOL;
```

Parameters

UsbControlTransfer	Accesses the USB Device through USB Control Transfer Pipe. See the **UsbControlTransfer()** function description.
UsbBulkTransfer	Accesses the USB Device through USB Bulk Transfer Pipe. See the **UsbBulkTransfer()** function description.
UsbAsyncInterruptTransfer	Nonblock USB interrupt transfer. See the **UsbAsyncInterruptTransfer()** function description.
UsbSyncInterruptTransfer	Accesses the USB Device through USB Synchronous Interrupt Transfer Pipe. See the **UsbSyncInterruptTransfer()** function description.

UsbIsochronousTransfer	Accesses the USB Device through USB Isochronous Transfer Pipe. See the **UsbIsochronousTransfer()** function description.
UsbAsyncIsochronousTransfer	Nonblock USB isochronous transfer. See the **UsbAsyncIsochronousTransfer()** function description.
UsbGetDeviceDescriptor	Retrieves the device descriptor of a USB device. See the **UsbGetDeviceDescriptor()** function description.
UsbGetConfigDescriptor	Retrieves the activated configuration descriptor of a USB device. See the **UsbGetConfigDescriptor()** function description.
UsbGetInterfaceDescriptor	Retrieves the interface descriptor of a USB Controller. See the **UsbGetInterfaceDescriptor()** function description.
UsbGetEndpointDescriptor	Retrieves the endpoint descriptor of a USB Controller. See the **UsbGetEndpointDescriptor()** function description.
UsbGetStringDescriptor	Retrieves the string descriptor inside a USB Device. See the **UsbGetStringDescriptor()** function description.
UsbGetSupportedLanguages	Retrieves the array of languages that the USB device supports. See the **UsbGetSupportedLanguages()** function description.
UsbPortReset	Resets and reconfigures the USB controller. See the **UsbPortReset()** function description.

Description

The **EFI_USB_IO_PROTOCOL** provides four basic transfers types described in the *USB 1.1 Specification*. These include control transfer, interrupt transfer, bulk transfer and isochronous transfer. The **EFI_USB_IO_PROTOCOL** also provides some basic USB device/controller management and configuration interfaces. A USB device driver uses the services of this protocol to manage USB devices.

EFI_USB_IO_PROTOCOL.UsbControlTransfer()

Summary

This function is used to manage a USB device with a control transfer pipe. A control transfer is typically used to perform device initialization and configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_CONTROL_TRANSFER) (
  IN      EFI_USB_IO_PROTOCOL     *This,
  IN      EFI_USB_DEVICE_REQUEST  *Request,
  IN      EFI_USB_DATA_DIRECTION  Direction,
  IN      UINT32                  Timeout,
  IN OUT  VOID                    *Data        OPTIONAL,
  IN      UINTN                   DataLength   OPTIONAL,
  OUT     UINT32                  *Status
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
Request	A pointer to the USB device request that will be sent to the USB device. See “Related Definitions” below.
Direction	Indicates the data direction. See “Related Definitions” below for this type.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
Timeout	Indicating the transfer should be completed within this time frame. The units are in milliseconds. If *Timeout* is 0, then the caller must wait for the function to be completed until **EFI_SUCCESS** or **EFI_DEVICE_ERROR** is returned.
DataLength	The size, in bytes, of the data buffer specified by *Data*.
Status	A pointer to the result of the USB transfer.

Related Definitions

```
typedef enum {
  EfiUsbDataIn,
  EfiUsbDataOut,
  EfiUsbNoData
} EFI_USB_DATA_DIRECTION;

//
// Error code for USB Transfer Results
//
#define EFI_USB_NOERROR                 0x0000
#define EFI_USB_ERR_NOTEXECUTE          0x0001
#define EFI_USB_ERR_STALL               0x0002
#define EFI_USB_ERR_BUFFER              0x0004
#define EFI_USB_ERR_BABBLE              0x0008
#define EFI_USB_ERR_NAK                 0x0010
#define EFI_USB_ERR_CRC                 0x0020
#define EFI_USB_ERR_TIMEOUT             0x0040
#define EFI_USB_ERR_BITSTUFF            0x0080
#define EFI_USB_ERR_SYSTEM              0x0100

typedef struct {
  UINT8              RequestType;
  UINT8              Request;
  UINT16             Value;
  UINT16             Index;
  UINT16             Length;
} EFI_USB_DEVICE_REQUEST;
```

RequestType	The field identifies the characteristics of the specific request.
Request	This field specifies the particular request.
Value	This field is used to pass a parameter to USB device that is specific to the request.
Index	This field is also used to pass a parameter to USB device that is specific to the request.
Length	This field specifies the length of the data transferred during the second phase of the control transfer. If it is 0, then there is no data phase in this transfer.

Description

This function allows a USB device driver to communicate with the USB device through a Control Transfer. There are three control transfer types according to the data phase. If the *Direction* parameter is **EfiUsbNoData**, *Data* is **NULL**, and *DataLength* is 0, then no data phase exists for the control transfer. If the *Direction* parameter is **EfiUsbDataOut**, then *Data* specifies the data to be transmitted to the device, and *DataLength* specifies the number of bytes to transfer to the device. In this case there is an OUT DATA stage followed by a SETUP stage. If the *Direction* parameter is **EfiUsbDataIn**, then *Data* specifies the data that is received from the device, and *DataLength* specifies the number of bytes to receive from the device. In this case there is an IN DATA stage followed by a SETUP stage. After the USB transfer has completed successfully, **EFI_SUCCESS** is returned. If the transfer cannot be completed due to timeout, then **EFI_TIMEOUT** is returned. If an error other than timeout occurs during the USB transfer, then **EFI_DEVICE_ERROR** is returned and the detailed status code is returned in *Status*.

Status Code Returned

EFI_SUCCESS	The control transfer has been successfully executed.
EFI_INVALID_PARAMETER	The parameter *Direction* is not valid.
EFI_INVALID_PARAMETER	*Request* is **NULL**.
EFI-INVALID_PARAMETER	*Status* is **NULL**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_TIMEOUT	The control transfer fails due to timeout.
EFI_DEVICE_ERROR	The transfer failed. The transfer status is returned in *Status*.

EFI_USB_IO_PROTOCOL.UsbBulkTransfer()

Summary

This function is used to manage a USB device with the bulk transfer pipe. Bulk Transfers are typically used to transfer large amounts of data to/from USB devices.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_BULK_TRANSFER) (
  IN      EFI_USB_IO_PROTOCOL  *This,
  IN      UINT8                DeviceEndpoint,
  IN      OUT VOID             *Data,
  IN OUT  UINTN                *DataLength,
  IN      UINTN                Timeout,
  OUT     UINT32               *Status
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
DeviceEndpoint	The destination USB device endpoint to which the device request is being sent. *DeviceEndpoint* must be between 0x01 and 0x0F or between 0x81 and 0x8F, otherwise **EFI_INVALID_PARAMETER** is returned. If the endpoint is not a BULK endpoint, **EFI_INVALID_PARAMETER** is returned. The MSB of this parameter indicates the endpoint direction. The number "1" stands for an IN endpoint, and "0" stands for an OUT endpoint.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
DataLength	On input, the size, in bytes, of the data buffer specified by *Data*. On output, the number of bytes that were actually transferred.
Timeout	Indicating the transfer should be completed within this time frame. The units are in milliseconds. If *Timeout* is 0, then the caller must wait for the function to be completed until **EFI_SUCCESS** or **EFI_DEVICE_ERROR** is returned.
Status	This parameter indicates the USB transfer status.

Description

This function allows a USB device driver to communicate with the USB device through Bulk Transfer. The transfer direction is determined by the endpoint direction. If the USB transfer is successful, then **EFI_SUCCESS** is returned. If USB transfer cannot be completed within the *Timeout* frame, **EFI_TIMEOUT** is returned. If an error other than timeout occurs during the USB transfer, then **EFI_DEVICE_ERROR** is returned and the detailed status code will be returned in the *Status* parameter.

Status Code Returned

EFI_SUCCESS	The bulk transfer has been successfully executed.
EFI_INVALID_PARAMETER	If *DeviceEndpoint* is not valid.
EFI_INVALID_PARAMETER	*Data* is **NULL**.
EFI_INVALID_PARAMETER	*DataLength* is **NULL**.
EFI_INVALID_PARAMETER	*Status* is **NULL**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_TIMEOUT	The bulk transfer cannot be completed within *Timeout* timeframe.
EFI_DEVICE_ERROR	The transfer failed other than timeout, and the transfer status is returned in *Status*.

EFI_USB_IO_PROTOCOL.UsbAsyncInterruptTransfer()

Summary

This function is used to manage a USB device with an interrupt transfer pipe. An Asynchronous Interrupt Transfer is typically used to query a device's status at a fixed rate. For example, keyboard, mouse, and hub devices use this type of transfer to query their interrupt endpoints at a fixed rate.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_ASYNC_INTERRUPT_TRANSFER) (
  IN EFI_USB_IO_PROTOCOL               *This,
  IN UINT8                             DeviceEndpoint,
  IN BOOLEAN                           IsNewTransfer,
  IN UINTN                             PollingInterval   OPTIONAL,
  IN UINTN                             DataLength        OPTIONAL,
  IN EFI_ASYNC_USB_TRANSFER_CALLBACK   InterruptCallBack OPTIONAL,
  IN VOID                              *Context          OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
DeviceEndpoint	The destination USB device endpoint to which the device request is being sent. *DeviceEndpoint* must be between 0x01 and 0x0F or between 0x81 and 0x8F, otherwise **EFI_INVALID_PARAMETER** is returned. If the endpoint is not an INTERRUPT endpoint, **EFI_INVALID_PARAMETER** is returned. The MSB of this parameter indicates the endpoint direction. The number "1" stands for an IN endpoint, and "0" stands for an OUT endpoint.
IsNewTransfer	If **TRUE**, a new transfer will be submitted to USB controller. If **FALSE**, the interrupt transfer is deleted from the device's interrupt transfer queue.
PollingInterval	Indicates the periodic rate, in milliseconds, that the transfer is to be executed. This parameter is required when *IsNewTransfer* is **TRUE**. The value must be between 1 to 255, otherwise **EFI_INVALID_PARAMETER** is returned. The units are in milliseconds.
DataLength	Specifies the length, in bytes, of the data to be received from the USB device. This parameter is only required when *IsNewTransfer* is **TRUE**.
Context	Data passed to the *InterruptCallback* function. This is an optional parameter and may be **NULL**.

InterruptCallback	The Callback function. This function is called if the asynchronous interrupt transfer is completed. This parameter is required when *IsNewTransfer* is **TRUE**. See "Related Definitions" for the definition of this type.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI     * EFI_ASYNC_USB_TRANSFER_CALLBACK) (
  IN VOID     *Data,
  IN UINTN    DataLength,
  IN VOID     *Context,
  IN UINT32   Status
  );
```

Data	Data received or sent via the USB Asynchronous Transfer, if the transfer completed successfully.
DataLength	The length of *Data* received or sent via the Asynchronous Transfer, if transfer successfully completes.
Context	Data passed from **UsbAsyncInterruptTransfer()** request.
Status	Indicates the result of the asynchronous transfer.

Description

This function allows a USB device driver to communicate with a USB device with an Interrupt Transfer. Asynchronous Interrupt transfer is different than the other four transfer types because it is a nonblocking transfer. The interrupt endpoint is queried at a fixed rate, and the data transfer direction is always in the direction from the USB device towards the system.

If *IsNewTransfer* is **TRUE**, then an interrupt transfer is started at a fixed rate. The rate is specified by *PollingInterval*, the size of the receive buffer is specified by *DataLength*, and the callback function is specified by *InterruptCallback*.

If *IsNewTransfer* is **FALSE**, then the interrupt transfer is canceled.

Status Code Returned

EFI_SUCCESS	The asynchronous USB transfer request has been successfully executed.
EFI_DEVICE_ERROR	The asynchronous USB transfer request failed.

Examples

Below is an example of how an asynchronous interrupt transfer is used. The example shows how a USB Keyboard Device Driver can periodically receive data from interrupt endpoint.

```
EFI_USB_IO_PROTOCOL           *UsbIo;
EFI_STATUS                    Status;
USB_KEYBOARD_DEV              *UsbKeyboardDevice;
EFI_USB_INTERRUPT_CALLBACK    KeyboardHandle;

. . .
Status = UsbIo->UsbAsyncInterruptTransfer(
                    UsbIo,
             UsbKeyboardDevice->IntEndpointAddress,
             TRUE,
             UsbKeyboardDevice->IntPollingInterval,
             8,
             KeyboardHandler,
             UsbKeyboardDevice
             );
. . .

//
// The following is the InterruptCallback function. If there is any results got
// from Asynchoronous Interrupt Transfer, this function will be called.
//
EFI_STATUS
KeyboardHandler(
   IN VOID              *Data,
   IN UINTN             DataLength,
   IN   VOID            *Context,
   IN UINT32            Result
   )
{
   USB_KEYBOARD_DEV    *UsbKeyboardDevice;
   UINTN               I;

   if(EFI_ERROR(Result))
   {
      //
      // Something error during this transfer, just to some recovery work
      //
      . . .
      . . .
      return EFI_DEVICE_ERROR;
   }

   UsbKeyboardDevice = (USB_KEYBOARD_DEV *)Context;

   for(I = 0; I < DataLength; I++)
   {
      ParsedData(Data[I]);
      . . .
    }

    return EFI_SUCCESS;
}
```

EFI_USB_IO_PROTOCOL.UsbSyncInterruptTransfer()

Summary

This function is used to manage a USB device with an interrupt transfer pipe. The difference between **UsbAsyncInterruptTransfer()** and **UsbSyncInterruptTransfer()** is that the Synchronous interrupt transfer will only be executed one time. Once it returns, regardless of its status, the interrupt request will be deleted in the system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_SYNC_INTERRUPT_TRANSFER) (
  IN      EFI_USB_IO_PROTOCOL   *This,
  IN      UINT8                 DeviceEndpoint,
  IN OUT  VOID                  *Data,
  IN OUT  UINTN                 *DataLength,
  IN      UINTN                 Timeout,
  OUT     UINT32                *Status
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
DeviceEndpoint	The destination USB device endpoint to which the device request is being sent. *DeviceEndpoint* must be between 0x01 and 0x0F or between 0x81 and 0x8F, otherwise **EFI_INVALID_PARAMETER** is returned. If the endpoint is not an INTERRUPT endpoint, **EFI_INVALID_PARAMETER** is returned. The MSB of this parameter indicates the endpoint direction. The number "1" stands for an IN endpoint, and "0" stands for an OUT endpoint.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
DataLength	On input, then size, in bytes, of the buffer *Data*. On output, the amount of data actually transferred.
Timeout	The time out, in seconds, for this transfer. If *Timeout* is 0, then the caller must wait for the function to be completed until **EFI_SUCCESS** or **EFI_DEVICE_ERROR** is returned. If the transfer is not completed in this time frame, then **EFI_TIMEOUT** is returned.
Status	This parameter indicates the USB transfer status.

Description

This function allows a USB device driver to communicate with a USB device through a synchronous interrupt transfer. The **UsbSyncInterruptTransfer()** differs from **UsbAsyncInterruptTransfer()** described in the previous section in that it is a blocking transfer request. The caller must wait for the function return, either successfully or unsuccessfully.

Status Code Returned

EFI_SUCCESS	The sync interrupt transfer has been successfully executed.
EFI_INVALID_PARAMETER	The parameter *DeviceEndpoint* is not valid.
EFI_INVALID_PARAMETER	*Data* is **NULL.**
EFI_INVALID_PARAMETER	*DataLength* is **NULL**.
EFI_INVALID_PARAMETER	*Status* is **NULL**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_TIMEOUT	The transfer cannot be completed within *Timeout* timeframe.
EFI_DEVICE_ERROR	The transfer failed other than timeout, and the transfer status is returned in *Status*.

EFI_USB_IO_PROTOCOL.UsbIsochronousTransfer()

Summary

This function is used to manage a USB device with an isochronous transfer pipe. An Isochronous transfer is typically used to transfer streaming data.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_USB_IO_ISOCHRONOUS_TRANSFER) (
  IN      EFI_USB_IO_PROTOCOL  *This,
  IN      UINT8                DeviceEndpoint,
  IN OUT  VOID                 *Data,
  IN      UINTN                DataLength,
  OUT     UINT32               *Status
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
DeviceEndpoint	The destination USB device endpoint to which the device request is being sent. *DeviceEndpoint* must be between 0x01 and 0x0F or between 0x81 and 0x8F, otherwise **EFI_INVALID_PARAMETER** is returned. If the endpoint is not an ISOCHRONOUS endpoint, **EFI_INVALID_PARAMETER** is returned. The MSB of this parameter indicates the endpoint direction. The number "1" stands for an IN endpoint, and "0" stands for an OUT endpoint.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
DataLength	The size, in bytes, of the data buffer specified by *Data*.
Status	This parameter indicates the USB transfer status.

Description

This function allows a USB device driver to communicate with a USB device with an Isochronous Transfer. The type of transfer is different than the other types because the USB Bus Driver will not attempt to perform error recovery if transfer fails. If the USB transfer is completed successfully, then **EFI_SUCCESS** is returned. The isochronous transfer is designed to be completed within 1 USB frame time, if it cannot be completed, **EFI_TIMEOUT** is returned. If the transfer fails due to other reasons, then **EFI_DEVICE_ERROR** is returned and the detailed error status is returned in *Status*. If the data length exceeds the maximum payload per USB frame time, then it is this function's responsibility to divide the data into a set of smaller packets that fit into a USB frame time. If all the packets are transferred successfully, then **EFI_SUCCESS** is returned.

Status Code Returned

EFI_SUCCESS	The isochronous transfer has been successfully executed.
EFI_INVALID_PARAMETER	The parameter *DeviceEndpoint* is not valid.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_TIMEOUT	The transfer cannot be completed within the 1 USB frame time.
EFI_DEVICE_ERROR	The transfer failed due to the reason other than timeout, The error status is returned in *Status*.

EFI_USB_IO_PROTOCOL.UsbAsyncIsochronousTransfer()

Summary

This function is used to manage a USB device with an isochronous transfer pipe. An asynchronous Isochronous transfer is a nonblocking USB isochronous transfer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_ASYNC_ISOCHRONOUS_TRANSFER) (
  IN EFI_USB_IO_PROTOCOL              *This,
  IN UINT8                            DeviceEndpoint,
  IN OUT VOID                         *Data,
  IN UINTN                            DataLength,
  IN EFI_ASYNC_USB_TRANSFER_CALLBACK IsochronousCallBack,
  IN VOID                             *Context       OPTIONAL
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
DeviceEndpoint	The destination USB device endpoint to which the device request is being sent. *DeviceEndpoint* must be between 0x01 and 0x0F or between 0x81 and 0x8F, otherwise **EFI_INVALID_PARAMETER** is returned. If the endpoint is not an ISOCHRONOUS endpoint, **EFI_INVALID_PARAMETER** is returned. The MSB of this parameter indicates the endpoint direction. The number "1" stands for an IN endpoint, and "0" stands for an OUT endpoint.
Data	A pointer to the buffer of data that will be transmitted to USB device or received from USB device.
DataLength	Specifies the length, in bytes, of the data to be sent to or received from the USB device.
Context	Data passed to the **IsochronousCallback()** function. This is an optional parameter and may be **NULL**.
IsochronousCallback	The **IsochronousCallback()** function. This function is called if the requested isochronous transfer is completed. See the "Related Definitions" section of the **UsbAsyncInterruptTransfer()** function description.

Description

This is an asynchronous type of USB isochronous transfer. If the caller submits a USB isochronous transfer request through this function, this function will return immediately. When the isochronous transfer completes, the `IsochronousCallback()` function will be triggered, the caller can know the transfer results. If the transfer is successful, the caller can get the data received or sent in this callback function.

Status Code Returned

EFI_SUCCESS	The asynchronous isochronous transfer has been successfully submitted to the system.
EFI_INVALID_PARAMETER	The parameter *DeviceEndpoint* is not valid.
EFI_OUT_OF_RESOURCES	The request could not be submitted due to a lack of resources.

EFI_USB_IO_PROTOCOL.UsbGetDeviceDescriptor()

Summary

Retrieves the USB Device Descriptor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_GET_DEVICE_DESCRIPTOR) (
  IN    EFI_USB_IO_PROTOCOL          *This,
  OUT   EFI_USB_DEVICE_DESCRIPTOR    *DeviceDescriptor
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
DeviceDescriptor	A pointer to the caller allocated USB Device Descriptor. See "Related Definitions" for a detailed description.

Related Definitions

```
//
// See USB1.1 for detail descrption.
//
typedef struct {
  UINT8    Length;
  UINT8    DescriptorType;
  UINT16   BcdUSB;
  UINT8    DeviceClass;
  UINT8    DeviceSubClass;
  UINT8    DeviceProtocol;
  UINT8    MaxPacketSize0;
  UINT16   IdVendor;
  UINT16   IdProduct;
  UINT16   BcdDevice;
  UINT8    StrManufacturer;
  UINT8    StrProduct;
  UINT8    StrSerialNumber;
  UINT8    NumConfigurations;
} EFI_USB_DEVICE_DESCRIPTOR;
```

Description

This function is used to retrieve information about USB devices. This information includes the device class, subclass, and the number of configurations the USB device supports. If *DeviceDescriptor* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If the USB device descriptor is not found, then **EFI_NOT_FOUND** is returned. Otherwise, the device descriptor is returned in *DeviceDescriptor*, and **EFI_SUCCESS** is returned.

Status Code Returned

EFI_SUCCESS	The device descriptor was retrieved successfully.
EFI_INVALID_PARAMETER	*DeviceDescriptor* is NULL.
EFI_NOT_FOUND	The device descriptor was not found. The device may not be configured.

EFI_USB_IO_PROTOCOL.UsbGetConfigDescriptor()

Summary

Retrieves the USB Device Configuration Descriptor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_GET_CONFIG_DESCRIPTOR) (
  IN    EFI_USB_IO_PROTOCOL          *This,
  OUT   EFI_USB_CONFIG_DESCRIPTOR    *ConfigurationDescriptor
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
ConfigurationDescriptor	A pointer to the caller allocated USB Active Configuration Descriptor. See "Related Definitions" for a detailed description.

Related Definitions

```
//
// See USB1.1 for detail descrption.
//
typedef struct {
  UINT8    Length;
  UINT8    DescriptorType;
  UINT16   TotalLength;
  UINT8    NumInterfaces;
  UINT8    ConfigurationValue;
  UINT8    Configuration;
  UINT8    Attributes;
  UINT8    MaxPower;
} EFI_USB_CONFIG_DESCRIPTOR;
```

Description

This function is used to retrieve the active configuration that the USB device is currently using. If *ConfigurationDescriptor* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If the USB controller does not contain an active configuration, then **EFI_NOT_FOUND** is returned. Otherwise, the active configuration is returned in *ConfigurationDescriptor*, and **EFI_SUCCESS** is returned.

Status Code Returned

EFI_SUCCESS	The active configuration descriptor was retrieved successfully.
EFI_INVALID_PARAMETER	*ConfigurationDescriptor* is NULL.
EFI_NOT_FOUND	An active configuration descriptor cannot be found. The device may not be configured.

EFI_USB_IO_PROTOCOL.UsbGetInterfaceDescriptor()

Summary

Retrieves the Interface Descriptor for a USB Device Controller. As stated earlier, an interface within a USB device is equivalently to a USB Controller within the current configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_GET_INTERFACE_DESCRIPTOR) (
  IN    EFI_USB_IO_PROTOCOL              *This,
  OUT   EFI_USB_INTERFACE_DESCRIPTOR     *InterfaceDescriptor
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
InterfaceDescriptor	A pointer to the caller allocated USB Interface Descriptor within the configuration setting. See "Related Definitions" for a detailed description.

Related Definitions

```
//
// See USB1.1 for detail descrption.
//
typedef struct {
  UINT8   Length;
  UINT8   DescriptorType;
  UINT8   InterfaceNumber;
  UINT8   AlternateSetting;
  UINT8   NumEndpoints;
  UINT8   InterfaceClass;
  UINT8   InterfaceSubClass;
  UINT8   InterfaceProtocol;
  UINT8   Interface;
} EFI_USB_INTERFACE_DESCRIPTOR;
```

Description

This function is used to retrieve the interface descriptor for the USB controller. If *`InterfaceDescriptor`* is **`NULL`**, then **`EFI_INVALID_PARAMETER`** is returned. If the USB controller does not contain an interface descriptor, then **`EFI_NOT_FOUND`** is returned. Otherwise, the interface descriptor is returned in *`InterfaceDescriptor`*, and **`EFI_SUCCESS`** is returned.

Status Code Returned

EFI_SUCCESS	The interface descriptor retrieved successfully.
EFI_INVALID_PARAMETER	*`InterfaceDescriptor`* is **`NULL`**.
EFI_NOT_FOUND	The interface descriptor cannot be found. The device may not be correctly configured.

EFI_USB_IO_PROTOCOL.UsbGetEndpointDescriptor()

Summary

Retrieves an Endpoint Descriptor within a USB Controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_GET_ENDPOINT_DESCRIPTOR) (
  IN    EFI_USB_IO_PROTOCOL             *This,
  IN    UINT8                           EndpointIndex,
  OUT   EFI_USB_ENDPOINT_DESCRIPTOR     *EndpointDescriptor
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
EndpointIndex	Indicates which endpoint descriptor to retrieve. The valid range is 0..15.
EndpointDescriptor	A pointer to the caller allocated USB Endpoint Descriptor of a USB controller. See “Related Definitions” for a detailed description.

Related Definitions

```
//
// See USB1.1 for detail descrption.
//
typedef struct {
  UINT8    Length;
  UINT8    DescriptorType;
  UINT8    EndpointAddress;
  UINT8    Attributes;
  UINT16   MaxPacketSize;
  UINT8    Interval;
} EFI_USB_ENDPOINT_DESCRIPTOR;
```

Description

This function is used to retrieve an endpoint descriptor within a USB controller. If *EndpointIndex* is not in the range 0..15, then **EFI_INVALID_PARAMETER** is returned. If *EndpointDescriptor* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If the endpoint specified by *EndpointIndex* does not exist within the USB controller, then **EFI_NOT_FOUND** is returned. Otherwise, the endpoint descriptor is returned in *EndpointDescriptor*, and **EFI_SUCCESS** is returned.

Status Code Returned

EFI_SUCCESS	The endpoint descriptor was retrieved successfully.
EFI_INVALID_PARAMETER	*EndpointIndex* is not valid.
EFI_INVALID_PARAMETER	*EndpointDescriptor* is **NULL**.
EFI_NOT_FOUND	The endpoint descriptor cannot be found. The device may not be correctly configured.

Examples

The following code fragment shows how to retrieve all the endpoint descriptors from a USB controller.

```
EFI_USB_IO_PROTOCOL                *UsbIo;
EFI_USB_INTERFACE_DESCRIPTOR       InterfaceDesc;
EFI_USB_ENDPOINT_DESCRIPTOR        EndpointDesc;
UINTN                              Index;

Status = UsbIo->GetInterfaceDescriptor (
                  UsbIo,
                  &InterfaceDesc
                  );
. . .
for(Index = 0; Index < InterfaceDesc.NumEndpoints; Index++) {
  Status = UsbIo->GetEndpointDescriptor(
                    UsbIo,
                    Index,
                    &EndpointDesc
                    );
. . .
}
```

EFI_USB_IO_PROTOCOL.UsbGetStringDescriptor()

Summary

Retrieves a Unicode string stored in a USB Device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_GET_STRING_DESCRIPTOR) (
  IN    EFI_USB_IO_PROTOCOL    *This,
  IN    UINT16                 LangID,
  IN    UINT8                  StringID,
  OUT   CHAR16                 **String
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
LangID	The Language ID for the string being retrieved. See the **UsbGetSupportedLanguages()** function description for a more detailed description.
StringID	The ID of the string being retrieved.
String	A pointer to a buffer allocated by this function with **AllocatePool()** to store the string. If this function returns **EFI_SUCCESS**, it stores the string the caller wants to get. The caller should release the string buffer with **FreePool()** after the string is not used any more.

Description

This function is used to retrieve strings stored in a USB device. Strings are stored in a Unicode format. The string to retrieve is identified by a language and an identifier. The language is specified by *LangID*, and the identifier is specified by *StringID*. If the string is found, it is returned in *String*, and **EFI_SUCCESS** is returned. If the string cannot be found, then **EFI_NOT_FOUND** is returned. The string buffer is allocated by this function with **AllocatePool()**. The caller is responsible for calling **FreePool()** for *String* when it is no longer required.

Status Code Returned

EFI_SUCCESS	The string was retrieved successfully.
EFI_NOT_FOUND	The string specified by *LangID* and *StringID* was not found.
EFI_OUT_OF_RESOURCES	There are not enough resources to allocate the return buffer *String*.

EFI_USB_IO_PROTOCOL.UsbGetSupportedLanguages()

Summary

Retrieves all the language ID codes that the USB device supports.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_GET_SUPPORTED_LANGUAGES) (
  IN  EFI_USB_IO_PROTOCOL   *This,
  OUT UINT16                **LangIDTable,
  OUT UINT16                *TableSize
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.
LangIDTable	Language ID for the string the caller wants to get. This is a 16-bit ID defined by Microsoft. This buffer pointer is allocated and maintained by the USB Bus Driver, the caller should not modify its contents.
TableSize	The size, in bytes, of the table *LangIDTable*.

Description

Retrieves all the language ID codes that the USB device supports.

Status Code Returned

EFI_SUCCESS	The support languages were retrieved successfully.

EFI_USB_IO_PROTOCOL.UsbPortReset()

Summary

Resets and reconfigures the USB controller. This function will work for all USB devices except USB Hub Controllers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB_IO_PORT_RESET) (
  IN  EFI_USB_IO_PROTOCOL  *This
  );
```

Parameters

This	A pointer to the **EFI_USB_IO_PROTOCOL** instance. Type **EFI_USB_IO_PROTOCOL** is defined in Section 14.2.5.

Description

This function provides a reset mechanism by sending a RESET signal from the parent hub port. A reconfiguration process will happen (that includes setting the address and setting the configuration). This reset function does not change the bus topology. A USB hub controller cannot be reset using this function, because it would impact the downstream USB devices. So if the controller is a USB hub controller, then **EFI_INVALID_PARAMETER** is returned.

Status Code Returned

EFI_SUCCESS	The USB controller was reset.
EFI_INVALID_PARAMETER	If the controller specified by *This* is a USB hub.
EFI_DEVICE_ERROR	An error occurred during the reconfiguration process.

14.2.6 USB Device Paths

An EFI_USB_IO_PROTOCOL must be installed on a handle for its services to be available to USB device drivers. In addition to the EFI_USB_IO_PROTOCOL, an EFI_DEVICE_PATH must also be installed on the same handle. See Chapter 5 for a detailed description of the EFI_DEVICE_PATH.

14.2.6.1 USB Device Path Node

Below is the definition for USB Device Path Node.

```
#define MESSAGING_DEVICE_PATH          0x03
#define MSG_USB_DP                     0x05
typedef struct _USB_DEVICE_PATH {
  EFI_DEVICE_PATH  Header;
  UINT8            ParentPortNumber;
  UINT8            InterfaceNumber;
} USB_DEVICE_PATH;
```

Parameters

Header	Device Path Header for USB Device Path Node.
ParentPortNumber	The parent USB hub port which this controller is connected to.
InterfaceNumber	The USB Controller identifier within a USB device. Some USB device may have several interfaces (i.e. USB Controllers). The Interface Number is the identifier for the USB Controller.

14.2.6.2 USB Device Path Example

Table 14-4 shows an example device path for a USB controller on a desktop platform. This USB Controller is connected to the port 0 of the root hub, and its interface number is 0. The USB Host Controller is a PCI device whose PCI device number 0x1F and PCI function 0x02. So, the whole device path for this USB Controller consists an ACPI Device Path Node, a PCI Device Path Node, a USB Device Path Node and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

```
ACPI(PNP0A03,0)/PCI(1F|2)/USB(0,0).
```

Table 14-4. USB Device Path Examples

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x1F	PCI Function
0x11	0x01	0x02	PCI Device
0x12	0x01	0x03	**Generic Device Path Header** – Type Message Device Path
0x13	0x01	0x05	Sub type – USB
0x14	0x02	0x06	Length – 0x06 bytes
0x16	0x01	0x00	Parent Hub Port Number.
0x17	0x01	0x00	Controller Interface Number.
0x18	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x19	0x01	0xFF	Sub type – End of Entire Device Path
0x1A	0x02	0x04	Length – 0x04 bytes

Another example is a USB Controller (interface number 0) that is connected to port 3 of a USB Hub Controller (interface number 0), and this USB Hub Controller is connected to the port 1 of the root hub. The shorthand notation for this device path is:

`ACPI(PNP0A03,0)/PCI(1F|2)/USB(1,0)/USB(3,0).`

Table 14-5 shows the device path for this USB Controller.

Table 14-5. Another USB Device Path Example

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	**Generic Device Path Header** – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes.
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	**Generic Device Path Header** – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x1F	PCI Function
0x11	0x01	0x02	PCI Device
0x12	0x01	0x03	**Generic Device Path Header** – Type Message Device Path
0x13	0x01	0x05	Sub type – USB
0x14	0x02	0x06	Length – 0x06 bytes
0x16	0x01	0x01	Parent Hub Port Number.
0x17	0x01	0x00	Controller Interface Number.
0x18	0x01	0x03	**Generic Device Path Header** – Type Message Device Path
0x19	0x01	0x05	Sub type – USB
0x1A	0x02	0x06	Length – 0x06 bytes
0x1C	0x01	0x03	Parent Hub Port Number.
0x1D	0x01	0x00	Controller Interface Number.
0x1E	0x01	0xFF	**Generic Device Path Header** – Type End of Hardware Device Path
0x1F	0x01	0xFF	Sub type – End of Entire Device Path
0x20	0x02	0x04	Length – 0x04 bytes

15 Protocols - Network Support

15.1 EFI_SIMPLE_NETWORK Protocol

This section defines the Simple Network Protocol. This protocol provides a packet level interface to a network adapter.

EFI_SIMPLE_NETWORK Protocol

Summary

The **EFI_SIMPLE_NETWORK** protocol provides services to initialize a network interface, transmit packets, receive packets, and close a network interface.

GUID

```
#define EFI_SIMPLE_NETWORK_PROTOCOL \
     { A19832B9-AC25-11D3-9A2D-0090273fc14d }
```

Revision Number

```
#define EFI_SIMPLE_NETWORK_INTERFACE_REVISION    0x00010000
```

Protocol Interface Structure

```
typedef struct _EFI_SIMPLE_NETWORK_ {
     UINT64                                 Revision;
     EFI_SIMPLE_NETWORK_START               Start;
     EFI_SIMPLE_NETWORK_STOP                Stop;
     EFI_SIMPLE_NETWORK_INITIALIZE          Initialize;
     EFI_SIMPLE_NETWORK_RESET               Reset;
     EFI_SIMPLE_NETWORK_SHUTDOWN            Shutdown;
     EFI_SIMPLE_NETWORK_RECEIVE_FILTERS     ReceiveFilters;
     EFI_SIMPLE_NETWORK_STATION_ADDRESS     StationAddress;
     EFI_SIMPLE_NETWORK_STATISTICS          Statistics;
     EFI_SIMPLE_NETWORK_MCAST_IP_TO_MAC     MCastIpToMac;
     EFI_SIMPLE_NETWORK_NVDATA              NvData;
     EFI_SIMPLE_NETWORK_GET_STATUS          GetStatus;
     EFI_SIMPLE_NETWORK_TRANSMIT            Transmit;
     EFI_SIMPLE_NETWORK_RECEIVE             Receive;
     EFI_EVENT                              WaitForPacket;
     EFI_SIMPLE_NETWORK_MODE                *Mode;
} EFI_SIMPLE_NETWORK;
```

Parameters

Revision	Revision of the **EFI_SIMPLE_NETWORK** Protocol. All future revisions must be backwards compatible. If a future version is not backwards compatible it is not the same GUID.
Start	Prepares the network interface for further command operations. No other **EFI_SIMPLE_NETWORK** interface functions will operate until this call is made. See the **Start()** function description.
Stop	Stops further network interface command processing. No other **EFI_SIMPLE_NETWORK** interface functions will operate after this call is made until another **Start()** call is made. See the **Stop()** function description.
Initialize	Resets the network adapter and allocates the transmit and receive buffers. See the **Initialize()** function description.
Reset	Resets the network adapter and reinitializes it with the parameters provided in the previous call to **Initialize()**. See the **Reset()** function description.
Shutdown	Resets the network adapter and leaves it in a state that is safe for another driver to initialize. The memory buffers assigned in the **Initialize()** call are released. After this call, only the **Initialize()** or **Stop()** calls may be used. See the **Shutdown()** function description.
ReceiveFilters	Enables and disables the receive filters for the network interface and, if supported, manages the filtered multicast HW MAC (Hardware Media Access Control) address list. See the **ReceiveFilters()** function description.
StationAddress	Modifies or resets the current station address, if supported. See the **StationAddress()** function description.
Statistics	Collects statistics from the network interface and allows the statistics to be reset. See the **Statistics()** function description.
MCastIpToMac	Maps a multicast IP address to a multicast HW MAC address. See the **MCastIpToMac()** function description.
NvData	Reads and writes the contents of the NVRAM devices attached to the network interface. See the **NvData()** function description.
GetStatus	Reads the current interrupt status and the list of recycled transmit buffers from the network interface. See the **GetStatus()** function description.
Transmit	Places a packet in the transmit queue. See the **Transmit()** function description.

Receive	Retrieves a packet from the receive queue, along with the status flags that describe the packet type. See the **Receive()** function description.
WaitForPacket	Event used with **WaitForEvent()** to wait for a packet to be received.
Mode	Pointer to the **EFI_SIMPLE_NETWORK_MODE** data for the device. See "Related Definitions" below.

Related Definitions

```
//*******************************************************
// EFI_SIMPLE_NETWORK_MODE
//
// Note that the fields in this data structure are read-only and
// are updated by the code that produces the EFI_SIMPLE_NETWORK
// protocol functions.  All these fields must be discovered
// during driver initialization.
//*******************************************************
typedef struct {
     UINT32                State;
     UINT32                HwAddressSize;
     UINT32                MediaHeaderSize;
     UINT32                MaxPacketSize;
     UINT32                NvRamSize;
     UINT32                NvRamAccessSize;
     UINT32                ReceiveFilterMask;
     UINT32                ReceiveFilterSetting;
     UINT32                MaxMCastFilterCount;
     UINT32                MCastFilterCount;
     EFI_MAC_ADDRESS       MCastFilter[MAX_MCAST_FILTER_CNT];
     EFI_MAC_ADDRESS       CurrentAddress;
     EFI_MAC_ADDRESS       BroadcastAddress;
     EFI_MAC_ADDRESS       PermanentAddress;
     UINT8                 IfType;
     BOOLEAN               MacAddressChangeable;
     BOOLEAN               MultipleTxSupported;
     BOOLEAN               MediaPresentSupported;
     BOOLEAN               MediaPresent;
} EFI_SIMPLE_NETWORK_MODE;
```

State	Reports the current state of the network interface (see **EFI_SIMPLE_NETWORK_STATE** below). When an **EFI_SIMPLE_NETWORK** driver has initialized a network interface, it is left in the **EfiSimpleNetworkStopped** state.
HwAddressSize	The size, in bytes, of the network interface's HW address.
MediaHeaderSize	The size, in bytes, of the network interface's media header.

MaxPacketSize	The maximum size, in bytes, of the packets supported by the network interface.
NvRamSize	The size, in bytes, of the NVRAM device attached to the network interface. If an NVRAM device is not attached to the network interface, then this field will be zero. This value must be a multiple of *NvramAccessSize*.
NvRamAccessSize	The size that must be used for all NVRAM accesses. This means that the start address for NVRAM read and write operations, and the total length of those operations, must be a multiple of this value. The legal values for this field are 0, 1, 2, 4, 8. If the value is zero, then no NVRAM devices are attached to the network interface.
ReceiveFilterMask	The multicast receive filter settings supported by the network interface.
ReceiveFilterSetting	The current multicast receive filter settings. See "Bit Mask Values for *ReceiveFilterSetting*" below.
MaxMCastFilterCount	The maximum number of multicast address receive filters supported by the driver. If this value is zero, then the multicast address receive filters cannot be modified with ReceiveFilters(). This field may be less than **MAX_MCAST_FILTER_CNT** (see below).
MCastFilterCount	The current number of multicast address receive filters.
MCastFilter	Array containing the addresses of the current multicast address receive filters.
CurrentAddress	The current HW MAC address for the network interface.
BroadcastAddress	The current HW MAC address for broadcast packets.
PermanentAddress	The permanent HW MAC address for the network interface.
IfType	The interface type of the network interface. See RFC 1700, section "Number Hardware Type."
MacAddressChangeable	**TRUE** if the HW MAC address can be changed.
MultipleTxSupported	**TRUE** if the network interface can transmit more than one packet at a time.
MediaPresentSupported	**TRUE** if the presence of media can be determined; otherwise **FALSE**. If **FALSE**, *MediaPresent* cannot be used.
MediaPresent	**TRUE** if media are connected to the network interface; otherwise **FALSE**. This field is only valid immediately after calling **Initialize()**.

```
//*******************************************************
// EFI_SIMPLE_NETWORK_STATE
//*******************************************************
typedef enum {
     EfiSimpleNetworkStopped,
     EfiSimpleNetworkStarted,
     EfiSimpleNetworkInitialized,
     EfiSimpleNetworkMaxState
} EFI_SIMPLE_NETWORK_STATE;

//*******************************************************
// MAX_MCAST_FILTER_CNT
//*******************************************************
#define MAX_MCAST_FILTER_CNT               16

//*******************************************************
// Bit Mask Values for ReceiveFilterSetting.   bit mask values
//
// Note that all other bit values are reserved.
//*******************************************************
#define EFI_SIMPLE_NETWORK_RECEIVE_UNICAST                0x01
#define EFI_SIMPLE_NETWORK_RECEIVE_MULTICAST              0x02
#define EFI_SIMPLE_NETWORK_RECEIVE_BROADCAST              0x04
#define EFI_SIMPLE_NETWORK_RECEIVE_PROMISCUOUS            0x08
#define EFI_SIMPLE_NETWORK_RECEIVE_PROMISCUOUS_MULTICAST  0x10
```

Description

The **EFI_SIMPLE_NETWORK** protocol is used to initialize access to a network adapter. Once the network adapter has been initialized, the **EFI_SIMPLE_NETWORK** protocol provides services that allow packets to be transmitted and received. This provides a packet level interface that can then be used by higher level drivers to produce boot services like DHCP, TFTP, and MTFTP. In addition, this protocol can be used as a building block in a full UDP and TCP/IP implementation that can produce a wide variety of application level network interfaces. See the *Preboot Execution Environment (PXE) Specification* for more information.

Implementation Note

The underlying network hardware may only be able to access 4 GB (32-bits) of system memory. Any requests to transfer data to/from memory above 4 GB with 32-bit network hardware will be double-buffered (using intermediate buffers below 4 GB) and will reduce performance.

EFI_SIMPLE_NETWORK.Start()

Summary

Changes the state of a network interface from "stopped" to "started."

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_START) (
      IN EFI_SIMPLE_NETWORK        *This
      );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.

Description

This function starts a network interface. If the network interface was successfully started, then **EFI_SUCCESS** will be returned.

Status Codes Returned

EFI_SUCCESS	The network interface was started.
EFI_ALREADY_STARTED	The network interface is already in the started state.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.Stop()

Summary

Changes the state of a network interface from "started" to "stopped."

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_STOP) (
      IN EFI_SIMPLE_NETWORK        *This
      );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.

Description

This function stops an network interface. This call is only valid if the network interface is in the started state. If the network interface was successfully stopped, then **EFI_SUCCESS** will be returned.

Status Codes Returned

EFI_SUCCESS	The network interface was stopped.
EFI_NOT_STARTED	The network interface has not been started.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.Initialize()

Summary

Resets a network adapter and allocates the transmit and receive buffers required by the network interface; optionally, also requests allocation of additional transmit and receive buffers.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_INITIALIZE) (
     IN EFI_SIMPLE_NETWORK      *This,
     IN UINTN                   ExtraRxBufferSize    OPTIONAL,
     IN UINTN                   ExtraTxBufferSize    OPTIONAL
     );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
ExtraRxBufferSize	The size, in bytes, of the extra receive buffer space that the driver should allocate for the network interface. Some network interfaces will not be able to use the extra buffer, and the caller will not know if it is actually being used.
ExtraTxBufferSize	The size, in bytes, of the extra transmit buffer space that the driver should allocate for the network interface. Some network interfaces will not be able to use the extra buffer, and the caller will not know if it is actually being used.

Description

This function allocates the transmit and receive buffers required by the network interface. If this allocation fails, then **EFI_OUT_OF_RESOURCES** is returned. If the allocation succeeds and the network interface is successfully initialized, then **EFI_SUCCESS** will be returned.

Status Codes Returned

EFI_SUCCESS	The network interface was initialized.
EFI_NOT_STARTED	The network interface has not been started.
EFI_OUT_OF_RESOURCES	There was not enough memory for the transmit and receive buffers.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.Reset()

Summary

Resets a network adapter and reinitializes it with the parameters that were provided in the previous call to **Initialize()**.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_RESET) (
      IN EFI_SIMPLE_NETWORK       *This,
      IN BOOLEAN                  ExtendedVerification
      );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
ExtendedVerification	Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description

This function resets a network adapter and reinitializes it with the parameters that were provided in the previous call to **Initialize()**. The transmit and receive queues are emptied and all pending interrupts are cleared. Receive filters, the station address, the statistics, and the multicast-IP-to-HW MAC addresses are not reset by this call. If the network interface was successfully reset, then **EFI_SUCCESS** will be returned. If the driver has not been initialized, **EFI_DEVICE_ERROR** will be returned.

Status Codes Returned

EFI_SUCCESS	The network interface was reset.
EFI_NOT_STARTED	The network interface has not been started.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.Shutdown()

Summary

Resets a network adapter and leaves it in a state that is safe for another driver to initialize.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_SHUTDOWN) (
     IN EFI_SIMPLE_NETWORK          *This
     );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.

Description

This function releases the memory buffers assigned in the **Initialize()** call. Pending transmits and receives are lost, and interrupts are cleared and disabled. After this call, only the **Initialize()** and **Stop()** calls may be used. If the network interface was successfully shutdown, then **EFI_SUCCESS** will be returned. If the driver has not been initialized, **EFI_DEVICE_ERROR** will be returned.

Status Codes Returned

EFI_SUCCESS	The network interface was shutdown.
EFI_NOT_STARTED	The network interface has not been started.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.ReceiveFilters()

Summary

Manages the multicast receive filters of a network interface.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_RECEIVE_FILTERS) (
      IN EFI_SIMPLE_NETWORK             *This,
      IN UINT32                         Enable,
      IN UINT32                         Disable,
      IN BOOLEAN                        ResetMCastFilter,
      IN UINTN                          MCastFilterCnt    OPTIONAL,
      IN EFI_MAC_ADDRESS               *MCastFilter       OPTIONAL,
      );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
Enable	A bit mask of receive filters to enable on the network interface.
Disable	A bit mask of receive filters to disable on the network interface.
ResetMCastFilter	Set to **TRUE** to reset the contents of the multicast receive filters on the network interface to their default values.
MCastFilterCnt	Number of multicast HW MAC addresses in the new *MCastFilter* list. This value must be less than or equal to the *MCastFilterCnt* field of **EFI_SIMPLE_NETWORK_MODE**. This field is optional if *ResetMCastFilter* is **TRUE**.
MCastFilter	A pointer to a list of new multicast receive filter HW MAC addresses. This list will replace any existing multicast HW MAC address list. This field is optional if *ResetMCastFilter* is **TRUE**.

Description

This function modifies the current receive filter mask on a network interface. The bits set in *Enable* are set on the current receive filter mask. The bits set in *Disable* are cleared from the current receive filter mask. If the same bit is set in both *Enable* and *Disable*, then the bit will be disabled. The receive filter mask is updated on the network interface, and the new receive filter mask can be read from the *ReceiveFilterSetting* field of **EFI_SIMPLE_NETWORK_MODE**. If an attempt is made to enable a bit that is not supported on the network interface, then **EFI_INVALID_PARAMETER** will be returned. The *ReceiveFilterMask* field of **EFI_SIMPLE_NETWORK_MODE** specifies the supported receive filters settings. See "Bit Mask Values for *ReceiveFilterSetting*" in "Related Definitions" in Section 15.1 for the list of the supported receive filter bit mask values.

If *ResetMCastFilter* is **TRUE**, then the multicast receive filter list on the network interface will be reset to the default multicast receive filter list. If *ResetMCastFilter* is **FALSE**, and this network interface allows the multicast receive filter list to be modified, then the *MCastFilterCnt* and *MCastFilter* are used to update the current multicast receive filter list. The modified receive filter list settings can be found in the *MCastFilter* field of **EFI_SIMPLE_NETWORK_MODE**. If the network interface does not allow the multicast receive filter list to be modified, then **EFI_INVALID_PARAMETER** will be returned. If the driver has not been initialized, **EFI_DEVICE_ERROR** will be returned.

If the receive filter mask and multicast receive filter list have been successfully updated on the network interface, **EFI_SUCCESS** will be returned.

Status Codes Returned

EFI_SUCCESS	The multicast receive filter list was updated.
EFI_NOT_STARTED	The network interface has not been started.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.StationAddress()

Summary

Modifies or resets the current station address, if supported.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_STATION_ADDRESS) (
      IN EFI_SIMPLE_NETWORK            *This,
      IN BOOLEAN                       Reset,
      IN EFI_MAC_ADDRESS               *New      OPTIONAL
      );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
Reset	Flag used to reset the station address to the network interface's permanent address.
New	New station address to be used for the network interface.

Description

This function modifies or resets the current station address of a network interface, if supported. If *Reset* is **TRUE**, then the current station address is set to the network interface's permanent address. If *Reset* is **FALSE**, and the network interface allows its station address to be modified, then the current station address is changed to the address specified by *New*. If the network interface does not allow its station address to be modified, then **EFI_INVALID_PARAMETER** will be returned. If the station address is successfully updated on the network interface, **EFI_SUCCESS** will be returned. If the driver has not been initialized, **EFI_DEVICE_ERROR** will be returned.

Status Codes Returned

EFI_SUCCESS	The network interface's station address was updated.
EFI_NOT_STARTED	The network interface has not been started.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.Statistics()

Summary

Resets or collects the statistics on a network interface.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_STATISTICS) (
     IN EFI_SIMPLE_NETWORK       *This,
     IN BOOLEAN                  Reset,
     IN OUT UINTN                *StatisticsSize       OPTIONAL,
     OUT EFI_NETWORK_STATISTICS  *StatisticsTable      OPTIONAL
     );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
Reset	Set to **TRUE** to reset the statistics for the network interface.
StatisticsSize	On input the size, in bytes, of *StatisticsTable*. On output the size, in bytes, of the resulting table of statistics.
StatisticsTable	A pointer to the **EFI_NETWORK_STATISTICS** structure that contains the statistics. Type **EFI_NETWORK_STATISTICS** is defined in “Related Definitions” below.

Related Definitions

```
//*******************************************************
// EFI_NETWORK_STATISTICS
//
// Any statistic value that is -1 is not available
// on the device and is to be ignored.
//*******************************************************
typedef struct {
     UINT64                RxTotalFrames;
     UINT64                RxGoodFrames;
     UINT64                RxUndersizeFrames;
     UINT64                RxOversizeFrames;
     UINT64                RxDroppedFrames;
     UINT64                RxUnicastFrames;
     UINT64                RxBroadcastFrames;
     UINT64                RxMulticastFrames;
     UINT64                RxCrcErrorFrames;
     UINT64                RxTotalBytes;
     UINT64                TxTotalFrames;
```

```
    UINT64                TxGoodFrames;
    UINT64                TxUndersizeFrames;
    UINT64                TxOversizeFrames;
    UINT64                TxDroppedFrames;
    UINT64                TxUnicastFrames;
    UINT64                TxBroadcastFrames;
    UINT64                TxMulticastFrames;
    UINT64                TxCrcErrorFrames;
    UINT64                TxTotalBytes;
    UINT64                Collisions;
    UINT64                UnsupportedProtocol;
} EFI_NETWORK_STATISTICS;
```

RxTotalFrames	Total number of frames received. Includes frames with errors and dropped frames.
RxGoodFrames	Number of valid frames received and copied into receive buffers.
RxUndersizeFrames	Number of frames below the minimum length for the media. This would be less than 64 for Ethernet.
RxOversizeFrames	Number of frames longer than the maximum length for the media. This would be greater than 1500 for Ethernet.
RxDroppedFrames	Valid frames that were dropped because receive buffers were full.
RxUnicastFrames	Number of valid unicast frames received and not dropped.
RxBroadcastFrames	Number of valid broadcast frames received and not dropped.
RxMulticastFrames	Number of valid multicast frames received and not dropped.
RxCrcErrorFrames	Number of frames with CRC or alignment errors.
RxTotalBytes	Total number of bytes received. Includes frames with errors and dropped frames.
TxTotalFrames	Total number of frames transmitted. Includes frames with errors and dropped frames.
TxGoodFrames	Number of valid frames transmitted and copied into receive buffers.
TxUndersizeFrames	Number of frames below the minimum length for the media. This would be less than 64 for Ethernet.
TxOversizeFrames	Number of frames longer than the maximum length for the media. This would be greater than 1500 for Ethernet.
TxDroppedFrames	Valid frames that were dropped because receive buffers were full.
TxUnicastFrames	Number of valid unicast frames transmitted and not dropped.
TxBroadcastFrames	Number of valid broadcast frames transmitted and not dropped.

TxMulticastFrames	Number of valid multicast frames transmitted and not dropped.
TxCrcErrorFrames	Number of frames with CRC or alignment errors.
TxTotalBytes	Total number of bytes transmitted. Includes frames with errors and dropped frames.
Collisions	Number of collisions detected on this subnet.
UnsupportedProtocol	Number of frames destined for unsupported protocol.

Description

This function resets or collects the statistics on a network interface. If the size of the statistics table specified by *StatisticsSize* is not big enough for all the statistics that are collected by the network interface, then a partial buffer of statistics is returned in *StatisticsTable*, *StatisticsSize* is set to the size required to collect all the available statistics, and **EFI_BUFFER_TOO_SMALL** is returned.

If *StatisticsSize* is big enough for all the statistics, then *StatisticsTable* will be filled, *StatisticsSize* will be set to the size of the returned *StatisticsTable* structure, and **EFI_SUCCESS** is returned. If the driver has not been initialized, **EFI_DEVICE_ERROR** will be returned.

If *Reset* is **FALSE**, and both *StatisticsSize* and *StatisticsTable* are **NULL**, then no operations will be performed, and **EFI_SUCCESS** will be returned.

If *Reset* is **TRUE**, then all of the supported statistics counters on this network interface will be reset to zero.

Status Codes Returned

EFI_SUCCESS	The statistics were collected from the network interface.
EFI_NOT_STARTED	The network interface has not been started.
EFI_BUFFER_TOO_SMALL	The *Statistics* buffer was too small. The current buffer size needed to hold the statistics is returned in *StatisticsSize*.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.MCastIPtoMAC()

Summary

Converts a multicast IP address to a multicast HW MAC address.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_MCAST_IP_TO_MAC) (
      IN EFI_SIMPLE_NETWORK                     *This,
      IN BOOLEAN                                IPv6,
      IN EFI_IP_ADDRESS                         *IP,
      OUT EFI_MAC_ADDRESS                       *MAC
      );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
IPv6	Set to **TRUE** if the multicast IP address is IPv6 [RFC 2460]. Set to **FALSE** if the multicast IP address is IPv4 [RFC 791].
IP	The multicast IP address that is to be converted to a multicast HW MAC address.
MAC	The multicast HW MAC address that is to be generated from *IP*.

Description

This function converts a multicast IP address to a multicast HW MAC address for all packet transactions. If the mapping is accepted, then **EFI_SUCCESS** will be returned.

Status Codes Returned

EFI_SUCCESS	The multicast IP address was mapped to the multicast HW MAC address.
EFI_NOT_STARTED	The network interface has not been started.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.NvData()

Summary

Performs read and write operations on the NVRAM device attached to a network interface.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_NVDATA) (
     IN EFI_SIMPLE_NETWORK        *This
     IN BOOLEAN                   ReadWrite,
     IN UINTN                     Offset,
     IN UINTN                     BufferSize,
     IN OUT VOID                  *Buffer
     );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
ReadWrite	**TRUE** for read operations, **FALSE** for write operations.
Offset	Byte offset in the NVRAM device at which to start the read or write operation. This must be a multiple of *NvRamAccessSize* and less than *NvRamSize*. (See **EFI_SIMPLE_NETWORK_MODE**)
BufferSize	The number of bytes to read or write from the NVRAM device. This must also be a multiple of *NvramAccessSize*.
Buffer	A pointer to the data buffer.

Description

This function performs read and write operations on the NVRAM device attached to a network interface. If *ReadWrite* is TRUE, a read operation is performed. If *ReadWrite* is FALSE, a write operation is performed.

Offset specifies the byte offset at which to start either operation. *Offset* must be a multiple of *NvRamAccessSize* , and it must have a value between zero and *NvRamSize*.

BufferSize specifies the length of the read or write operation. *BufferSize* must also be a multiple of *NvRamAccessSize*, and *Offset* + *BufferSize* must not exceed *NvRamSize*.

If any of the above conditions is not met, then **EFI_INVALID_PARAMETER** will be returned.

If all the conditions are met and the operation is "read," the NVRAM device attached to the network interface will be read into *Buffer* and **EFI_SUCCESS** will be returned. If this is a write operation, the contents of *Buffer* will be used to update the contents of the NVRAM device attached to the network interface and **EFI_SUCCESS** will be returned.

Status Codes Returned

EFI_SUCCESS	The NVRAM access was performed.
EFI_NOT_STARTED	The network interface has not been started.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.GetStatus()

Summary

Reads the current interrupt status and recycled transmit buffer status from a network interface.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_GET_STATUS) (
      IN EFI_SIMPLE_NETWORK        *This,
      OUT UINT32                   *InterruptStatus    OPTIONAL,
      OUT VOID                     **TxBuf             OPTIONAL
      );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
InterruptStatus	A pointer to the bit mask of the currently active interrupts (see "Related Definitions"). If this is **NULL**, the interrupt status will not be read from the device. If this is not **NULL**, the interrupt status will be read from the device. When the interrupt status is read, it will also be cleared. Clearing the transmit interrupt does not empty the recycled transmit buffer array.
TxBuf	Recycled transmit buffer address. The network interface will not transmit if its internal recycled transmit buffer array is full. Reading the transmit buffer does not clear the transmit interrupt. If this is **NULL**, then the transmit buffer status will not be read. If there are no transmit buffers to recycle and *TxBuf* is not **NULL**, * *TxBuf* will be set to **NULL**.

Related Definitions

```
//*******************************************************
// Interrupt Bit Mask Settings for InterruptStatus.
// Note that all other bit values are reserved.
//*******************************************************
#define EFI_SIMPLE_NETWORK_RECEIVE_INTERRUPT          0x01
#define EFI_SIMPLE_NETWORK_TRANSMIT_INTERRUPT         0x02
#define EFI_SIMPLE_NETWORK_COMMAND_INTERRUPT          0x04
#define EFI_SIMPLE_NETWORK_SOFTWARE_INTERRUPT         0x08
```

Description

This function gets the current interrupt and recycled transmit buffer status from the network interface. The interrupt status is returned as a bit mask in *`InterruptStatus`*. If *`InterruptStatus`* is **`NULL`**, the interrupt status will not be read. If *`TxBuf`* is not **`NULL`**, a recycled transmit buffer address will be retrieved. If a recycled transmit buffer address is returned in *`TxBuf`*, then the buffer has been successfully transmitted, and the status for that buffer is cleared. If the status of the network interface is successfully collected, **`EFI_SUCCESS`** will be returned. If the driver has not been initialized, **`EFI_DEVICE_ERROR`** will be returned.

Status Codes Returned

EFI_SUCCESS	The status of the network interface was retrieved.
EFI_NOT_STARTED	The network interface has not been started.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.Transmit()

Summary

Places a packet in the transmit queue of a network interface.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_TRANSMIT) (
     IN EFI_SIMPLE_NETWORK             *This
     IN UINTN                          HeaderSize,
     IN UINTN                          BufferSize,
     IN VOID                           *Buffer,
     IN EFI_MAC_ADDRESS                *SrcAddr      OPTIONAL,
     IN EFI_MAC_ADDRESS                *DestAddr     OPTIONAL,
     IN UINT16                         *Protocol     OPTIONAL,
     );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
HeaderSize	The size, in bytes, of the media header to be filled in by the **Transmit()** function. If *HeaderSize* is nonzero, then it must be equal to *This->Mode->MediaHeaderSize* and the *DestAddr* and *Protocol* parameters must not be **NULL**.
BufferSize	The size, in bytes, of the entire packet (media header and data) to be transmitted through the network interface.
Buffer	A pointer to the packet (media header followed by data) to be transmitted. This parameter cannot be **NULL**. If *HeaderSize* is zero, then the media header in *Buffer* must already be filled in by the caller. If *HeaderSize* is nonzero, then the media header will be filled in by the **Transmit()** function.
SrcAddr	The source HW MAC address. If *HeaderSize* is zero, then this parameter is ignored. If *HeaderSize* is nonzero and *SrcAddr* is **NULL**, then *This->Mode->CurrentAddress* is used for the source HW MAC address.
DestAddr	The destination HW MAC address. If *HeaderSize* is zero, then this parameter is ignored.
Protocol	The type of header to build. If *HeaderSize* is zero, then this parameter is ignored. See RFC 1700, section "Ether Types," for examples.

Description

This function places the packet specified by *Header* and *Buffer* on the transmit queue. If *HeaderSize* is nonzero and *HeaderSize* is not equal to *This->Mode->MediaHeaderSize*, then **EFI_INVALID_PARAMETER** will be returned. If *BufferSize* is less than *This->Mode->MediaHeaderSize*, then **EFI_BUFFER_TOO_SMALL** will be returned. If *Buffer* is **NULL**, then **EFI_INVALID_PARAMETER** will be returned. If *HeaderSize* is nonzero and *DestAddr* or *Protocol* is **NULL**, then **EFI_INVALID_PARAMETER** will be returned. If the transmit engine of the network interface is busy, then **EFI_NOT_READY** will be returned. If this packet can be accepted by the transmit engine of the network interface, the packet contents specified by *Buffer* will be placed on the transmit queue of the network interface, and **EFI_SUCCESS** will be returned. **GetStatus()** can be used to determine when the packet has actually been transmitted. The contents of the *Buffer* must not be modified until the packet has actually been transmitted.

The **Transmit()** function performs nonblocking I/O. A caller who wants to perform blocking I/O, should call **Transmit()**, and then **GetStatus()** until the transmitted buffer shows up in the recycled transmit buffer.

If the driver has not been initialized, **EFI_DEVICE_ERROR** will be returned.

Status Codes Returned

EFI_SUCCESS	The packet was placed on the transmit queue.
EFI_NOT_STARTED	The network interface has not been started.
EFI_NOT_READY	The network interface is too busy to accept this transmit request.
EFI_BUFFER_TOO_SMALL	The *BufferSize* parameter is too small.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

EFI_SIMPLE_NETWORK.Receive()

Summary

Receives a packet from a network interface.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_RECEIVE) (
      IN EFI_SIMPLE_NETWORK            *This
      OUT UINTN                        *HeaderSize    OPTIONAL,
      IN OUT UINTN                     *BufferSize,
      OUT VOID                         *Buffer,
      OUT EFI_MAC_ADDRESS              *SrcAddr       OPTIONAL,
      OUT EFI_MAC_ADDRESS              *DestAddr      OPTIONAL,
      OUT UINT16                       *Protocol      OPTIONAL
      );
```

Parameters

This	A pointer to the **EFI_SIMPLE_NETWORK** instance.
HeaderSize	The size, in bytes, of the media header received on the network interface. If this parameter is **NULL**, then the media header size will not be returned.
BufferSize	On entry, the size, in bytes, of *Buffer*. On exit, the size, in bytes, of the packet that was received on the network interface.
Buffer	A pointer to the data buffer to receive both the media header and the data.
SrcAddr	The source HW MAC address. If this parameter is **NULL**, the HW MAC source address will not be extracted from the media header.
DestAddr	The destination HW MAC address. If this parameter is **NULL**, the HW MAC destination address will not be extracted from the media header.
Protocol	The media header type. If this parameter is **NULL**, then the protocol will not be extracted from the media header. See RFC 1700 section “Ether Types” for examples.

Description

This function retrieves one packet from the receive queue of a network interface. If there are no packets on the receive queue, then **EFI_NOT_READY** will be returned. If there is a packet on the receive queue, and the size of the packet is smaller than *BufferSize*, then the contents of the packet will be placed in *Buffer*, and *BufferSize* will be updated with the actual size of the packet. In addition, if *SrcAddr*, *DestAddr*, and *Protocol* are not **NULL**, then these values will be extracted from the media header and returned. **EFI_SUCCESS** will be returned if a packet was successfully received. If *BufferSize* is smaller than the received packet, then the size of the receive packet will be placed in *BufferSize* and **EFI_BUFFER_TOO_SMALL** will be returned. If the driver has not been initialized, **EFI_DEVICE_ERROR** will be returned.

Status Codes Returned

EFI_SUCCESS	The received data was stored in *Buffer*, and *BufferSize* has been updated to the number of bytes received.
EFI_NOT_STARTED	The network interface has not been started.
EFI_NOT_READY	No packets have been received on the network interface.
EFI_BUFFER_TOO_SMALL	*BufferSize* is too small for the received packets. *BufferSize* has been updated to the required size.
EFI_INVALID_PARAMETER	One or more of the parameters has an unsupported value.
EFI_DEVICE_ERROR	The command could not be sent to the network interface.
EFI_UNSUPPORTED	This function is not supported by the network interface.

15.2 NETWORK_INTERFACE_IDENTIFIER Protocol

This is an optional protocol that is used to describe details about the software layer that is used to produce the Simple Network Protocol. This protocol is only required if the underlying network interface is 16-bit UNDI, 32/64-bit S/W UNDI, or H/W UNDI. It is used to obtain type and revision information about the underlying network interface.

An instance of the Network Interface Identifier protocol must be created for each physical external network interface that is controlled by the !PXE structure. The !PXE structure is defined in the 32/64-bit UNDI Specification in Appendix E.

EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL

Summary

An optional protocol that is used to describe details about the software layer that is used to produce the Simple Network Protocol.

GUID

```
#define EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL \
     { E18541CD-F755-4f73-928D-643C8A79B229 }
```

Revision Number

```
#define EFI_NETWORK_INTERFACE_IDENTIFIER_INTERFACE_REVISION \
     0x00010000
```

Protocol Interface Structure

```
typedef struct {
     UINT64          Revision;
     UINT64          Id;
     UINT64          ImageAddr;
     UINT32          ImageSize;
     CHAR8           StringId[4];
     UINT8           Type;
     UINT8           MajorVer;
     UINT8           MinorVer;
     BOOLEAN         Ipv6Supported;
     UINT8           IfNum;
} EFI_NETWORK_INTERFACE_IDENTIFIER_INTERFACE;
```

Parameters

Revision	The revision of the **EFI_NETWORK_INTERFACE_IDENTIFIER** protocol.
Id	Address of the first byte of the identifying structure for this network interface. This is only valid when the network interface is started (see **EFI_SIMPLE_NETWORK_PROTOCOL.Start()**). When the network interface is not started, this field is set to zero. **16-bit UNDI and 32/64-bit S/W UNDI:** *Id* contains the address of the first byte of the copy of the **!PXE** structure in the relocated UNDI code segment. See the *Preboot Execution Environment (PXE) Specification* and Appendix E. **H/W UNDI:** *Id* contains the address of the **!PXE** structure.
ImageAddr	Address of the unrelocated network interface image. **16-bit UNDI:** *ImageAddr* is the address of the PXE option ROM image in upper memory. **32/64-bit S/W UNDI:** *ImageAddr* is the address of the unrelocated S/W UNDI image. **H/W UNDI:** *ImageAddr* contains zero.
ImageSize	Size of unrelocated network interface image. **16-bit UNDI:** *ImageSize* is the size of the PXE option ROM image in upper memory. **32/64-bit S/W UNDI:** *ImageSize* is the size of the unrelocated S/W UNDI image. **H/W UNDI:** *ImageSize* contains zero.
StringId	A four-character ASCII string that is sent in the class identifier field of option 60 in DHCP. For a *Type* of **EfiNetworkInterfaceUndi**, this field is “UNDI.”

Type	Network interface type. This will be set to one of the values in **EFI_NETWORK_INTERFACE_TYPE** (see "Related Definitions" below).
MajorVer	Major version number. **16-bit UNDI:** *MajorVer* comes from the third byte of the **UNDIRev** field in the **UNDI ROM ID** structure. Refer to the *Preboot Execution Environment (PXE) Specification*. **32/64-bit S/W UNDI and H/W UNDI:** *MajorVer* comes from the **Major** field in the **!PXE** structure. See Appendix E.
MinorVer	Minor version number. **16-bit UNDI:** *MinorVer* comes from the second byte of the **UNDIRev** field in the **UNDI ROM ID** structure. Refer to the *Preboot Execution Environment (PXE) Specification*. **32/64-bit S/W UNDI and H/W UNDI:** *MinorVer* comes from the **Minor** field in the **!PXE** structure. See Appendix E.
Ipv6Supported	**TRUE** if the network interface supports IPv6; otherwise **FALSE**.
IfNum	The network interface number that is being identified by this Network Interface Identifier Protocol. This field must be less than or equal to the IFcnt field in the !PXE structure.

Related Definitions

```
//*******************************************************
// EFI_NETWORK_INTERFACE_TYPE
//*******************************************************
typedef enum {
     EfiNetworkInterfaceUndi = 1
} EFI_NETWORK_INTERFACE_TYPE;
```

Description

The **EFI_NETWORK_INTERFACE_IDENTIFIER** Protocol is used by the **EFI_PXE_BASE_CODE** Protocol and OS loaders to identify the type of the underlying network interface and to locate its initial entry point.

15.3 PXE Base Code Protocol

This section defines the Preboot Execution Environment (PXE) Base Code protocol, which is used to access PXE-compatible devices for network access and network booting. More information about PXE can be found in the *Preboot Execution Environment (PXE) Specification* at: ftp://download.intel.com/ial/wfm/pxespec.pdf.

EFI_PXE_BASE_CODE Protocol

Summary

The **EFI_PXE_BASE_CODE** protocol is used to control PXE-compatible devices. The features of these devices are defined in the *Preboot Execution Environment (PXE) Specification*. An **EFI_PXE_BASE_CODE** protocol will be layered on top of an **EFI_SIMPLE_NETWORK** protocol in order to perform packet level transactions. The **EFI_PXE_BASE_CODE** handle also supports the **LOAD_FILE** protocol. This provides a clean way to obtain control from the boot manager if the boot path is from the remote device.

GUID

```
#define EFI_PXE_BASE_CODE_PROTOCOL \
     { 03C4E603-AC28-11d3-9A2D-0090273FC14D }
```

Revision Number

```
#define EFI_PXE_BASE_CODE_INTERFACE_REVISION     0x00010000
```

Protocol Interface Structure

```
typedef struct {
     UINT64                                      Revision;
     EFI_PXE_BASE_CODE_START                     Start;
     EFI_PXE_BASE_CODE_STOP                      Stop;
     EFI_PXE_BASE_CODE_DHCP                      Dhcp;
     EFI_PXE_BASE_CODE_DISCOVER                  Discover;
     EFI_PXE_BASE_CODE_MTFTP                     Mtftp;
     EFI_PXE_BASE_CODE_UDP_WRITE                 UdpWrite;
     EFI_PXE_BASE_CODE_UDP_READ                  UdpRead;
     EFI_PXE_BASE_CODE_SET_IP_FILTER             SetIpFilter;
     EFI_PXE_BASE_CODE_ARP                       Arp;
     EFI_PXE_BASE_CODE_SET_PARAMETERS            SetParameters;
     EFI_PXE_BASE_CODE_SET_STATION_IP            SetStationIp;
     EFI_PXE_BASE_CODE_SET_PACKETS               SetPackets;
     EFI_PXE_BASE_CODE_MODE                      *Mode;
} EFI_PXE_BASE_CODE;
```

Parameters

Revision	The revision of the **EFI_PXE_BASE_CODE** Protocol. All future revisions must be backwards compatible. If a future version is not backwards compatible it is not the same GUID.
Start	Starts the PXE Base Code Protocol. Mode structure information is not valid and no other Base Code Protocol functions will operate until the Base Code is started. See the **Start()** function description.
Stop	Stops the PXE Base Code Protocol. Mode structure information is unchanged by this function. No Base Code Protocol functions will operate until the Base Code is restarted. See the **Stop()** function description.
Dhcp	Attempts to complete a DHCPv4 D.O.R.A. (discover / offer / request / acknowledge) or DHCPv6 S.A.R.R (solicit / advertise / request / reply) sequence. See the **Dhcp()** function description.
Discover	Attempts to complete the PXE Boot Server and/or boot image discovery sequence. See the **Discover()** function description.
Mtftp	Performs TFTP and MTFTP services. See the **Mtftp()** function description.
UdpWrite	Writes a UDP packet to the network interface. See the **UdpWrite()** function description.
UdpRead	Reads a UDP packet from the network interface. See the **UdpRead()** function description.
SetIpFilter	Updates the IP receive filters of the network device. See the **SetIpFilter()** function description.
Arp	Uses the ARP protocol to resolve a MAC address. See the **Arp()** function description.
SetParameters	Updates the parameters that affect the operation of the PXE Base Code Protocol. See the **SetParameters()** function description.
SetStationIp	Updates the station IP address and subnet mask values. See the **SetStationIp()** function description.
SetPackets	Updates the contents of the cached DHCP and Discover packets. See the **SetPackets()** function description.
Mode	Pointer to the **EFI_PXE_BASE_CODE_MODE** data for this device. The **EFI_PXE_BASE_CODE_MODE** structure is defined in “Related Definitions” below.

Related Definitions

```
//*******************************************************
// Maximum ARP and Route Entries
//*******************************************************
#define EFI_PXE_BASE_CODE_MAX_ARP_ENTRIES        8
#define EFI_PXE_BASE_CODE_MAX_ROUTE_ENTRIES      8

//*******************************************************
// EFI_PXE_BASE_CODE_MODE
//
// The data values in this structure are read-only and
// are updated by the code that produces the EFI_PXE_BASE_CODE
// protocol functions.
//*******************************************************
typedef struct {
     BOOLEAN                         Started;
     BOOLEAN                         Ipv6Available;
     BOOLEAN                         Ipv6Supported;
     BOOLEAN                         UsingIpv6;
     BOOLEAN                         BisSupported;
     BOOLEAN                         BisDetected;
     BOOLEAN                         AutoArp;
     BOOLEAN                         SendGUID;
     BOOLEAN                         DhcpDiscoverValid;
     BOOLEAN                         DhcpAckReceivd;
     BOOLEAN                         ProxyOfferReceived;
     BOOLEAN                         PxeDiscoverValid;
     BOOLEAN                         PxeReplyReceived;
     BOOLEAN                         PxeBisReplyReceived;
     BOOLEAN                         IcmpErrorReceived;
     BOOLEAN                         TftpErrorReceived;
     BOOLEAN                         MakeCallbacks;
     UINT8                           TTL;
     UINT8                           ToS;
     EFI_IP_ADDRESS                  StationIp;
     EFI_IP_ADDRESS                  SubnetMask;
     EFI_PXE_BASE_CODE_PACKET        DhcpDiscover;
     EFI_PXE_BASE_CODE_PACKET        DhcpAck;
     EFI_PXE_BASE_CODE_PACKET        ProxyOffer;
     EFI_PXE_BASE_CODE_PACKET        PxeDiscover;
     EFI_PXE_BASE_CODE_PACKET        PxeReply;
     EFI_PXE_BASE_CODE_PACKET        PxeBisReply;
```

```
        EFI_PXE_BASE_CODE_IP_FILTER       IpFilter;
        UINT32                            ArpCacheEntries;
        EFI_PXE_BASE_CODE_ARP_ENTRY
             ArpCache[EFI_PXE_BASE_CODE_MAX_ARP_ENTRIES];
        UINT32                            RouteTableEntries;
        EFI_PXE_BASE_CODE_ROUTE_ENTRY
             RouteTable[EFI_PXE_BASE_CODE_MAX_ROUTE_ENTRIES];
        EFI_PXE_BASE_CODE_ICMP_ERROR      IcmpError;
        EFI_PXE_BASE_CODE_TFTP_ERROR      TftpError;
} EFI_PXE_BASE_CODE_MODE;
```

Started	**TRUE** if this device has been started by calling **Start()**. This field is set to **TRUE** by the **Start()** function and to **FALSE** by the **Stop()** function.
Ipv6Available	**TRUE** if the Simple Network Protocol being used supports IPv6.
Ipv6Supported	**TRUE** if this PXE Base Code Protocol implementation supports IPv6.
UsingIpv6	**TRUE** if this device is currently using IPv6. This field is set by the **Start()** function.
BisSupported	**TRUE** if this PXE Base Code implementation supports Boot Integrity Services (BIS). This field is set by the **Start()** function.
BisDetected	**TRUE** if this device and the platform support Boot Integrity Services (BIS). This field is set by the **Start()** function.
AutoArp	**TRUE** for automatic ARP packet generation; **FALSE** otherwise. This field is initialized to **TRUE** by **Start()** and can be modified with the **SetParameters()** function.
SendGUID	This field is used to change the Client Hardware Address (chaddr) field in the DHCP and Discovery packets. Set to **TRUE** to send the SystemGuid (if one is available). Set to **FALSE** to send the client NIC MAC address. This field is initialized to **FALSE** by **Start()** and can be modified with the **SetParameters()** function.
DhcpDiscoverValid	This field is initialized to **FALSE** by the **Start()** function and set to **TRUE** when the **Dhcp()** function completes successfully. When **TRUE**, the *DhcpDiscover* field is valid. This field can also be changed by the **SetPackets()** function.

DhcpAckReceived	This field is initialized to **FALSE** by the **Start()** function and set to **TRUE** when the **Dhcp()** function completes successfully. When **TRUE**, the *DhcpAck* field is valid. This field can also be changed by the **SetPackets()** function.
ProxyOfferReceived	This field is initialized to **FALSE** by the **Start()** function and set to **TRUE** when the **Dhcp()** function completes successfully and a proxy DHCP offer packet was received. When **TRUE**, the *ProxyOffer* packet field is valid. This field can also be changed by the **SetPackets()** function.
PxeDiscoverValid	When **TRUE**, the *PxeDiscover* packet field is valid. This field is set to **FALSE** by the **Start()** and **Dhcp()** functions, and can be set to **TRUE** or **FALSE** by the **Discover()** and **SetPackets()** functions.
PxeReplyReceived	When **TRUE,** the *PxeReply* packet field is valid. This field is set to **FALSE** by the **Start()** and **Dhcp()** functions, and can be set to **TRUE** or **FALSE** by the **Discover()** and **SetPackets()** functions.
PxeBisReplyReceived	When **TRUE**, the *PxeBisReply* packet field is valid. This field is set to **FALSE** by the **Start()** and **Dhcp()** functions, and can be set to **TRUE** or **FALSE** by the **Discover()** and **SetPackets()** functions.
IcmpErrorReceived	Indicates whether the *IcmpError* field has been updated. This field is reset to **FALSE** by the **Start()**, **Dhcp()**, **Discover()**, **Mtftp()**, **UdpRead()**, **UdpWrite()** and **Arp()** functions. If an ICMP error is received, this field will be set to **TRUE** after the *IcmpError* field is updated.
TftpErrorReceived	Indicates whether the *TftpError* field has been updated. This field is reset to **FALSE** by the **Start()** and **Mtftp()** functions. If a TFTP error is received, this field will be set to **TRUE** after the *TftpError* field is updated.
MakeCallbacks	When **FALSE**, callbacks will not be made. When **TRUE**, make callbacks to the PXE Base Code Callback Protocol. This field is reset to **FALSE** by the **Start()** function if the PXE Base Code Callback Protocol is not available. It is reset to **TRUE** by the **Start()** function if the PXE Base Code Callback Protocol is available.
TTL	The "time to live" field of the IP header. This field is initialized to **DEFAULT_TTL** (See "Related Definitions") by the **Start()** function and can be modified by the **SetParameters()** function.

ToS	The type of service field of the IP header. This field is initialized to **DEFAULT_ToS** (See "Related Definitions") by **Start()**, and can be modified with the **SetParameters()** function.
StationIp	The device's current IP address. This field is initialized to a zero address by **Start()**. This field is set when the **Dhcp()** function completes successfully. This field can also be set by the **SetStationIp()** function. This field must be set to a valid IP address by either **Dhcp()** or **SetStationIp()** before the **Discover()**, **Mtftp()**, **UdpRead()**, **UdpWrite()**, or **Arp()** functions are called.
SubnetMask	The device's current subnet mask. This field is initialized to a zero address by the **Start()** function. This field is set when the **Dhcp()** function completes successfully. This field can also be set by the **SetStationIp()** function. This field must be set to a valid subnet mask by either **Dhcp()** or **SetStationIp()** before the **Discover()**, **Mtftp()**, **UdpRead()**, **UdpWrite()**, or **Arp()** functions are called.
DhcpDiscover	Cached DHCP Discover packet. This field is zero-filled by the **Start()** function, and is set when the **Dhcp()** function completes successfully. The contents of this field can replaced by the **SetPackets()** function.
DhcpAck	Cached DHCP Ack packet. This field is zero-filled by the **Start()** function, and is set when the **Dhcp()** function completes successfully. The contents of this field can be replaced by the **SetPackets()** function.
ProxyOffer	Cached Proxy Offer packet. This field is zero-filled by the **Start()** function, and is set when the **Dhcp()** function completes successfully. The contents of this field can be replaced by the **SetPackets()** function.
PxeDiscover	Cached PXE Discover packet. This field is zero-filled by the **Start()** function, and is set when the **Discover()** function completes successfully. The contents of this field can be replaced by the **SetPackets()** function.
PxeReply	Cached PXE Reply packet. This field is zero-filled by the **Start()** function, and is set when the **Discover()** function completes successfully. The contents of this field can be replaced by the **SetPackets()** function.
PxeBisReply	Cached PXE BIS Reply packet. This field is zero-filled by the **Start()** function, and is set when the **Discover()** function completes successfully. This field can be replaced by the **SetPackets()** function.

IpFilter	The current IP receive filter settings. The receive filter is disabled and the number of IP receive filters is set to zero by the **Start()** function, and is set by the **SetIpFilter()** function.
ArpCacheEntries	The number of valid entries in the ARP cache. This field is reset to zero by the **Start()** function.
ArpCache	Array of cached ARP entries.
RouteTableEntries	The number of valid entries in the current route table. This field is reset to zero by the **Start()** function.
RouteTable	Array of route table entries.
IcmpError	ICMP error packet. This field is updated when an ICMP error is received and is undefined until the first ICMP error is received. This field is zero-filled by the **Start()** function.
TftpError	TFTP error packet. This field is updated when a TFTP error is received and is undefined until the first TFTP error is received. This field is zero-filled by the **Start()** function.

```
//*******************************************************
// EFI_PXE_BASE_CODE_UDP_PORT
//*******************************************************
typedef UINT16 EFI_PXE_BASE_CODE_UDP_PORT;

//*******************************************************
// EFI_IPv4_ADDRESS and EFI_IPv6_ADDRESS
//*******************************************************
typedef struct {
     UINT8                 Addr[4];
} EFI_IPv4_ADDRESS;

typedef struct {
     UINT8                 Addr[16];
} EFI_IPv6_ADDRESS;

//*******************************************************
// EFI_IP_ADDRESS
//*******************************************************
typedef union {
     UINT32                Addr[4];
     EFI_IPv4_ADDRESS      v4;
     EFI_IPv6_ADDRESS      v6;
} EFI_IP_ADDRESS;
```

```
//*******************************************************
// EFI_MAC_ADDRESS
//*******************************************************
typedef struct {
     UINT8                 Addr[32];
} EFI_MAC_ADDRESS;
```

DHCP Packet Data Types

This section defines the data types for DHCP packets, ICMP error packets, and TFTP error packets. All of these are byte-packed data structures.

NOTE

All the multibyte fields in these structures are stored in network order.

```
//*******************************************************
// EFI_PXE_BASE_CODE_DHCPV4_PACKET
//*******************************************************
typedef struct {
     UINT8                 BootpOpcode;
     UINT8                 BootpHwType;
     UINT8                 BootpHwAddrLen;
     UINT8                 BootpGateHops;
     UINT32                BootpIdent;
     UINT16                BootpSeconds;
     UINT16                BootpFlags;
     UINT8                 BootpCiAddr[4];
     UINT8                 BootpYiAddr[4];
     UINT8                 BootpSiAddr[4];
     UINT8                 BootpGiAddr[4];
     UINT8                 BootpHwAddr[16];
     UINT8                 BootpSrvName[64];
     UINT8                 BootpBootFile[128];
     UINT32                DhcpMagik;
     UINT8                 DhcpOptions[56];
} EFI_PXE_BASE_CODE_DHCPV4_PACKET;

// TBD in EFI v1.1
// typedef struct {
// } EFI_PXE_BASE_CODE_DHCPV6_PACKET;
```

```
//*******************************************************
// EFI_PXE_BASE_CODE_PACKET
//*******************************************************
typedef union {
     UINT64                          Alignment;
     UINT8                           Raw[1472];
     EFI_PXE_BASE_CODE_DHCPV4_PACKET Dhcpv4;
     // EFI_PXE_BASE_CODE_DHCPV6_PACKET    Dhcpv6;
} EFI_PXE_BASE_CODE_PACKET;

//*******************************************************
// EFI_PXE_BASE_CODE_ICMP_ERROR
//*******************************************************
typedef struct {
     UINT8                      Type;
     UINT8                      Code;
     UINT16                     Checksum;
     union {
          UINT32                reserved;
          UINT32                Mtu;
          UINT32                Pointer;
          struct {
               UINT16           Identifier;
               UINT16           Sequence;
          } Echo;
     } u;
UINT8                           Data[494];
} EFI_PXE_BASE_CODE_ICMP_ERROR;

//*******************************************************
// EFI_PXE_BASE_CODE_TFTP_ERROR
//*******************************************************
typedef struct {
     UINT8                      ErrorCode;
     CHAR8                      ErrorString[127];
} EFI_PXE_BASE_CODE_TFTP_ERROR;
```

IP Receive Filter Settings

This section defines the data types for IP receive filter settings.

```
#define EFI_PXE_BASE_CODE_MAX_IPCNT        8

//*******************************************************
// EFI_PXE_BASE_CODE_IP_FILTER
//*******************************************************
typedef struct {
     UINT8                    Filters;
     UINT8                    IpCnt;
     UINT16                   reserved;
     EFI_IP_ADDRESS           IpList[EFI_PXE_BASE_CODE_MAX_IPCNT];
} EFI_PXE_BASE_CODE_IP_FILTER;

#define EFI_PXE_BASE_CODE_IP_FILTER_STATION_IP             0x0001
#define EFI_PXE_BASE_CODE_IP_FILTER_BROADCAST              0x0002
#define EFI_PXE_BASE_CODE_IP_FILTER_PROMISCUOUS            0x0004
#define EFI_PXE_BASE_CODE_IP_FILTER_PROMISCUOUS_MULTICAST  0x0008
```

ARP Cache Entries

This section defines the data types for ARP cache entries, and route table entries.

```
//*******************************************************
// EFI_PXE_BASE_CODE_ARP_ENTRY
//*******************************************************
typedef struct {
     EFI_IP_ADDRESS                  IpAddr;
     EFI_MAC_ADDRESS                 MacAddr;
} EFI_PXE_BASE_CODE_ARP_ENTRY;

//*******************************************************
// EFI_PXE_BASE_CODE_ROUTE_ENTRY
//*******************************************************
typedef struct {
     EFI_IP_ADDRESS                  IpAddr;
     EFI_IP_ADDRESS                  SubnetMask;
     EFI_IP_ADDRESS                  GwAddr;
} EFI_PXE_BASE_CODE_ROUTE_ENTRY;
```

Filter Operations for UDP Read/Write Functions

This section defines the types of filter operations that can be used with the **UdpRead()** and **UdpWrite()** functions.

```
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_ANY_SRC_IP       0x0001
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_ANY_SRC_PORT     0x0002
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_ANY_DEST_IP      0x0004
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_ANY_DEST_PORT    0x0008
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_USE_FILTER       0x0010
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_MAY_FRAGMENT     0x0020
#define DEFAULT_TTL                                    4
#define DEFAULT_ToS                                    0
```

The following table defines values for the PXE DHCP and Bootserver Discover packet tags that are specific to the EFI environment. Complete definitions of all PXE tags are defined in Table 15-1 "PXE DHCP Options (Full List)," in the *PXE Specification*.

Table 15-1. PXE Tag Definitions for EFI

Tag Name	Tag #	Length	Data Field
Client Network Interface Identifier	94 [0x5E]	3 [0x03]	**Type (1), MajorVer (1), MinorVer (1)** Type is a one byte field that identifies the network interface that will be used by the downloaded program. Type is followed by two one byte version number fields, MajorVer and MinorVer. **Type** UNDI (1) = 0x01 **Versions** WfM-1.1a 16-bit UNDI: MajorVer = 0x02. MinorVer = 0x00 PXE-2.0 16-bit UNDI: MajorVer = 0x02, MinorVer = 0x01 32/64-bit UNDI & H/W UNDI: MajorVer = 0x03, MinorVer = 0x00
Client System Architecture	93 [0x5D]	2 [0x02]	**Type (2)** Type is a two byte, network order, field that identifies the processor and programming environment of the client system. **Types** IA x86 PC = 0x00 0x00 Itanium EFI PC = 0x00 0x02 IA-32 EFI PC = 0x00 0x06

continued

Table 15-1. PXE Tag Definitions for EFI (continued)

Tag Name	Tag #	Length	Data Field
Class Identifier	60 [0x3C]	32 [0x20]	**"PXEClient:Arch:xxxxx:UNDI:yyyzzz"** "PXEClient:…" is used to identify communication between PXE clients and servers. Information from tags 93 & 94 is embedded in the Class Identifier string. (The strings defined in this tag are case sensitive and must not be NULL-terminated.) xxxxx = ASCII representation of Client System Architecture. yyyzzz = ASCII representation of Client Network Interface Identifier version numbers MajorVer(yyy) and MinorVer(zzz). **Example** "PXEClient:Arch:00002:UNDI:00300" identifies an IA64 PC w/ 32/64-bit UNDI

Description

The basic mechanisms and flow for remote booting in EFI are identical to the remote boot functionality described in detail in the *PXE Specification*. However, the actual execution environment, linkage, and calling conventions are replaced and enhanced for the EFI environment.

The DHCP Option for the Client System Architecture is used to inform the DHCP server if the client is an EFI environment in an IA-32 or Itanium-based system. The server may use this information to provide default images if it does not have a specific boot profile for the client.

A handle that supports **EFI_PXE_BASE_CODE** protocol is required to support the **LOAD_FILE_Protocol** protocol. The **LOAD_FILE_Protocol** protocol function **LoadFile()** is used by the firmware to load files from devices that do not support file system type accesses. Specifically, the firmware's boot manager invokes **LoadFile()** with *BootPolicy* being **TRUE** when attempting to boot from the device. The firmware then loads and transfers control to the downloaded PXE boot image. Once the remote image is successfully loaded, it may utilize the **EFI_PXE_BASE_CODE** interfaces, or even the **EFI_SIMPLE_NETWORK** interfaces, to continue the remote process.

EFI_PXE_BASE_CODE.Start()

Summary

Enables the use of the PXE Base Code Protocol functions.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_START) (
      IN EFI_PXE_BASE_CODE        *This,
      IN BOOLEAN                  UseIpv6
      );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
UseIpv6	Specifies the type of IP addresses that are to be used during the session that is being started. Set to **TRUE** for IPv6 addresses, and **FALSE** for IPv4 addresses.

Description

This function enables the use of the PXE Base Code Protocol functions. If the *Started* field of the **EFI_PXE_BASE_CODE_MODE** structure is already **TRUE**, then **EFI_ALREADY_STARTED** will be returned. If *UseIpv6* is **TRUE**, then IPv6 formatted addresses will be used in this session. If *UseIpv6* is **FALSE**, then IPv4 formatted addresses will be used in this session. If *UseIpv6* is **TRUE**, and the *Ipv6Supported* field of the **EFI_PXE_BASE_CODE_MODE** structure is **FALSE**, then **EFI_UNSUPPORTED** will be returned. If there is not enough memory or other resources to start the PXE Base Code Protocol, then **EFI_OUT_OF_RESOURCES** will be returned. Otherwise, the PXE Base Code Protocol will be started, and all of the fields of the **EFI_PXE_BASE_CODE_MODE** structure will be initialized as follows:

Started	Set to **TRUE**.
Ipv6Supported	Unchanged.
Ipv6Available	Unchanged.
UsingIpv6	Set to *UseIpv6*.
BisSupported	Unchanged.
BisDetected	Unchanged.
AutoArp	Set to **TRUE**.
SendGUID	Set to **FALSE**.
TTL	Set to **DEFAULT_TTL**.
ToS	Set to **DEFAULT_ToS**.

DhcpCompleted	Set to **FALSE**.
ProxyOfferReceived	Set to **FALSE**.
StationIp	Set to an address of all zeros.
SubnetMask	Set to a subnet mask of all zeros.
DhcpDiscover	Zero-filled.
DhcpAck	Zero-filled.
ProxyOffer	Zero-filled.
PxeDiscoverValid	Set to **FALSE**.
PxeDiscover	Zero-filled.
PxeReplyValid	Set to **FALSE**.
PxeReply	Zero-filled.
PxeBisReplyValid	Set to **FALSE**.
PxeBisReply	Zero-filled.
IpFilter	Set the *Filters* field to 0 and the *IpCnt* field to 0.
ArpCacheEntries	Set to 0.
ArpCache	Zero-filled.
RouteTableEntries	Set to 0.
RouteTable	Zero-filled.
IcmpErrorReceived	Set to **FALSE**.
IcmpError	Zero-filled.
TftpErroReceived	Set to **FALSE**.
TftpError	Zero-filled.
MakeCallbacks	Set to **TRUE** if the PXE Base Code Callback Protocol is available. Set to **FALSE** if the PXE Base Code Callback Protocol is not available.

Status Codes Returned

EFI_SUCCESS	The PXE Base Code Protocol was started.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_UNSUPPORTED	*UseIpv6* is **TRUE**, but the *Ipv6Supported* field of the **EFI_PXE_BASE_CODE_MODE** structure is **FALSE**.
EFI_ALREADY_STARTED	The PXE Base Code Protocol is already in the started state.
EFI_DEVICE_ERROR	The network device encountered an error during this operation.
EFI_OUT_OF_RESOURCES	Could not allocate enough memory or other resources to start the PXE Base Code Protocol.

EFI_PXE_BASE_CODE.Stop()

Summary

Disables the use of the PXE Base Code Protocol functions.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_STOP) (
      IN EFI_PXE_BASE_CODE        *This
      );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.

Description

This function stops all activity on the network device. All the resources allocated in **Start()** are released, the *Started* field of the **EFI_PXE_BASE_CODE_MODE** structure is set to **FALSE** and **EFI_SUCCESS** is returned. If the *Started* field of the **EFI_PXE_BASE_CODE_MODE** structure is already **FALSE**, then **EFI_NOT_STARTED** will be returned.

Status Codes Returned

EFI_SUCCESS	The PXE Base Code Protocol was stopped.
EFI_NOT_STARTED	The PXE Base Code Protocol is already in the stopped state.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_DEVICE_ERROR	The network device encountered an error during this operation.

EFI_PXE_BASE_CODE.Dhcp()

Summary

Attempts to complete a DHCPv4 D.O.R.A. (discover / offer / request / acknowledge) or DHCPv6 S.A.R.R (solicit / advertise / request / reply) sequence.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_DHCP) (
      IN EFI_PXE_BASE_CODE        *This,
      IN BOOLEAN                  SortOffers
      );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
SortOffers	**TRUE** if the offers received should be sorted. Set to **FALSE** to try the offers in the order that they are received.

Description

This function attempts to complete the DHCP sequence. If this sequence is completed, then **EFI_SUCCESS** is returned, and the *DhcpCompleted*, *ProxyOfferReceived*, *StationIp*, *SubnetMask*, *DhcpDiscover*, *DhcpAck*, and *ProxyOffer* fields of the **EFI_PXE_BASE_CODE_MODE** structure are filled in.

If *SortOffers* is **TRUE**, then the cached DHCP offer packets will be sorted before they are tried. If *SortOffers* is **FALSE**, then the cached DHCP offer packets will be tried in the order in which they are received. Please see the *Preboot Execution Environment (PXE) Specification* for additional details on the implementation of DHCP.

This function can take at least 31 seconds to timeout and return control to the caller. If the DHCP sequence does not complete, then **EFI_TIMEOUT** will be returned.

If the Callback Protocol does not return **EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE**, then the DHCP sequence will be stopped and **EFI_ABORTED** will be returned.

Status Codes Returned

EFI_SUCCESS	Valid DHCP has completed.
EFI_NOT_STARTED	The PXE Base Code Protocol is in the stopped state.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_DEVICE_ERROR	The network device encountered an error during this operation.
EFI_OUT_OF_RESOURCES	Could not allocate enough memory to complete the DHCP Protocol.
EFI_ABORTED	The callback function aborted the DHCP Protocol.
EFI_TIMEOUT	The DHCP Protocol timed out.
EFI_ICMP_ERROR	The DHCP Protocol generated an ICMP error.
EFI_NO_RESPONSE	Valid PXE offer was not received.

EFI_PXE_BASE_CODE.Discover()

Summary

Attempts to complete the PXE Boot Server and/or boot image discovery sequence.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_DISCOVER) (
     IN EFI_PXE_BASE_CODE                      *This,
     IN UINT16                                 Type,
     IN UINT16                                 *Layer,
     IN BOOLEAN                                UseBis,
     IN EFI_PXE_BASE_CODE_DISCOVER_INFO        *Info    OPTIONAL
     );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
Type	The type of bootstrap to perform. See "Related Definitions" below.
Layer	Pointer to the boot server layer number to discover, which must be **PXE_BOOT_LAYER_INITIAL** when a new server type is being discovered. This is the only layer type that will perform multicast and broadcast discovery. All other layer types will only perform unicast discovery. If the boot server changes *Layer*, then the new *Layer* will be returned.
UseBis	**TRUE** if Boot Integrity Services are to be used. False otherwise.
Info	Pointer to a data structure that contains additional information on the type of discovery operation that is to be performed. If this field is **NULL**, then the contents of the cached *DhcpAck* and *ProxyOffer* packets will be used.

Related Definitions

```
//*******************************************************
// Bootstrap Types
//*******************************************************
#define EFI_PXE_BASE_CODE_BOOT_TYPE_BOOTSTRAP         0
#define EFI_PXE_BASE_CODE_BOOT_TYPE_MS_WINNT_RIS      1
#define EFI_PXE_BASE_CODE_BOOT_TYPE_INTEL_LCM         2
#define EFI_PXE_BASE_CODE_BOOT_TYPE_DOSUNDI           3
#define EFI_PXE_BASE_CODE_BOOT_TYPE_NEC_ESMPRO        4
#define EFI_PXE_BASE_CODE_BOOT_TYPE_IBM_WSoD          5
#define EFI_PXE_BASE_CODE_BOOT_TYPE_IBM_LCCM          6
#define EFI_PXE_BASE_CODE_BOOT_TYPE_CA_UNICENTER_TNG  7
```

```
#define EFI_PXE_BASE_CODE_BOOT_TYPE_HP_OPENVIEW       8
#define EFI_PXE_BASE_CODE_BOOT_TYPE_ALTIRIS_9         9
#define EFI_PXE_BASE_CODE_BOOT_TYPE_ALTIRIS_10        10
#define EFI_PXE_BASE_CODE_BOOT_TYPE_ALTIRIS_11        11
#define EFI_PXE_BASE_CODE_BOOT_TYPE_NOT_USED_12       12
#define EFI_PXE_BASE_CODE_BOOT_TYPE_REDHAT_INSTALL    13
#define EFI_PXE_BASE_CODE_BOOT_TYPE_REDHAT_BOOT       14
#define EFI_PXE_BASE_CODE_BOOT_TYPE_REMBO             15
#define EFI_PXE_BASE_CODE_BOOT_TYPE_BEOBOOT           16
//
// Values 17 through 32767 are reserved.
// Values 32768 through 65279 are for vendor use.
// Values 65280 through 65534 are reserved.
//
#define EFI_PXE_BASE_CODE_BOOT_TYPE_PXETEST           65535

#define EFI_PXE_BASE_CODE_BOOT_LAYER_MASK             0x7FFF
#define EFI_PXE_BASE_CODE_BOOT_LAYER_INITIAL          0x0000

//*******************************************************
// EFI_PXE_BASE_CODE_DISCOVER_INFO
//*******************************************************
typedef struct {
     BOOLEAN                          UseMCast;
     BOOLEAN                          UseBCast;
     BOOLEAN                          UseUCast;
     BOOLEAN                          MustUseList;
     EFI_IP_ADDRESS                   ServerMCastIp;
     UINT16                           IpCnt;
     EFI_PXE_BASE_CODE_SRVLIST        SrvList[IpCnt];
} EFI_PXE_BASE_CODE_DISCOVER_INFO;

//*******************************************************
// EFI_PXE_BASE_CODE_SRVLIST
//*******************************************************
typedef struct {
     UINT16                           Type;
     BOOLEAN                          AcceptAnyResponse;
     UINT8                            reserved;
     EFI_IP_ADDRESS                   IpAddr;
} EFI_PXE_BASE_CODE_SRVLIST;
```

Description

This function attempts to complete the PXE Boot Server and/or boot image discovery sequence. If this sequence is completed, then **EFI_SUCCESS** is returned, and the *PxeDiscoverValid*, *PxeDiscover*, *PxeReplyReceived*, and *PxeReply* fields of the **EFI_PXE_BASE_CODE_MODE** structure are filled in. If *UseBis* is **TRUE**, then the *PxeBisReplyReceived* and *PxeBisReply* fields of the **EFI_PXE_BASE_CODE_MODE** structure will also be filled in. If *UseBis* is **FALSE**, then *PxeBisReplyValid* will be set to **FALSE**.

In the structure referenced by parameter *Info*, the PXE Boot Server list, *SrvList []*, has two uses: It is the Boot Server IP address list used for unicast discovery (if the *UseUCast* field is **TRUE**), and it is the list used for Boot Server verification (if the *MustUseList* field is **TRUE**). Also, if the *MustUseList* field in that structure is **TRUE** and the *AcceptAnyResponse* field in the *SrvList []* array is **TRUE**, any Boot Server reply of that type will be accepted. If the *AcceptAnyResponse* field is **FALSE**, only responses from Boot Servers with matching IP addresses will be accepted.

This function can take at least 10 seconds to timeout and return control to the caller. If the Discovery sequence does not complete, then **EFI_TIMEOUT** will be returned. Please see the *Preboot Execution Environment (PXE) Specification* for additional details on the implementation of the Discovery sequence.

If the Callback Protocol does not return **EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE**, then the Discovery sequence is stopped and **EFI_ABORTED** will be returned.

Status Codes Returned

EFI_SUCCESS	The Discovery sequence has been completed.
EFI_NOT_STARTED	The PXE Base Code Protocol is in the stopped state.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_DEVICE_ERROR	The network device encountered an error during this operation.
EFI_OUT_OF_RESOURCES	Could not allocate enough memory to complete Discovery.
EFI_ABORTED	The callback function aborted the Discovery sequence.
EFI_TIMEOUT	The Discovery sequence timed out.
EFI_ICMP_ERROR	The Discovery sequence generated an ICMP error.

EFI_PXE_BASE_CODE.Mtftp()

Summary

Used to perform TFTP and MTFTP services.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_MTFTP) (
     IN EFI_PXE_BASE_CODE                 *This,
     IN EFI_PXE_BASE_CODE_TFTP_OPCODE     Operation,
     IN OUT VOID                          *BufferPtr, OPTIONAL
     IN BOOLEAN                           Overwrite,
     IN OUT UINTN                         *BufferSize,
     IN UINTN                             *BlockSize, OPTIONAL
     IN EFI_IP_ADDRESS                    *ServerIp,
     IN CHAR8                             *Filename, OPTIONAL
     IN EFI_PXE_BASE_CODE_MTFTP_INFO      *Info, OPTIONAL
     IN BOOLEAN                           DontUseBuffer
);
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
Operation	The type of operation to perform. See "Related Definitions" below for the list of operation types.
BufferPtr	A pointer to the data buffer. Ignored for read file if *DontUseBuffer* is **TRUE**.
Overwrite	Only used on write file operations. **TRUE** if a file on a remote server can be overwritten.
BufferSize	For get-file-size operations, **BufferSize* returns the size of the requested file. For read-file and write-file operations, this parameter is set to the size of the buffer specified by the *BufferPtr* parameter. For read-file operations, if **EFI_BUFFER_TOO_SMALL** is returned, **BufferSize* returns the size of the requested file.
BlockSize	The requested block size to be used during a TFTP transfer. This must be at least 512. If this field is **NULL**, then the largest block size supported by the implementation will be used.
ServerIp	The TFTP / MTFTP server IP address.
Filename	A Null-terminated ASCII string that specifies a directory name or a file name. This is ignored by MTFTP read directory.

Info	Pointer to the MTFTP information. This information is required to start or join a multicast TFTP session. It is also required to perform the "get file size" and "read directory" operations of MTFTP. See "Related Definitions" below for the description of this data structure.
DontUseBuffer	Set to **FALSE** for normal TFTP and MTFTP read file operation. Setting this to **TRUE** will cause TFTP and MTFTP read file operations to function without a receive buffer, and all of the received packets are passed to the Callback Protocol which is responsible for storing them. This field is only used by TFTP and MTFTP read file.

Related Definitions

```
//*******************************************************
// EFI_PXE_BASE_CODE_TFTP_OPCODE
//*******************************************************
typedef enum {
     EFI_PXE_BASE_CODE_TFTP_FIRST,
     EFI_PXE_BASE_CODE_TFTP_GET_FILE_SIZE,
     EFI_PXE_BASE_CODE_TFTP_READ_FILE,
     EFI_PXE_BASE_CODE_TFTP_WRITE_FILE,
     EFI_PXE_BASE_CODE_TFTP_READ_DIRECTORY,
     EFI_PXE_BASE_CODE_MTFTP_GET_FILE_SIZE,
     EFI_PXE_BASE_CODE_MTFTP_READ_FILE,
     EFI_PXE_BASE_CODE_MTFTP_READ_DIRECTORY,
     EFI_PXE_BASE_CODE_MTFTP_LAST
} EFI_PXE_BASE_CODE_TFTP_OPCODE;

//*******************************************************
// EFI_PXE_BASE_CODE_MTFTP_INFO
//*******************************************************
typedef struct {
     EFI_IP_ADDRESS                  MCastIp;
     EFI_PXE_BASE_CODE_UDP_PORT      CPort;
     EFI_PXE_BASE_CODE_UDP_PORT      SPort;
     UINT16                          ListenTimeout;
     UINT16                          TransmitTimeout;
} EFI_PXE_BASE_CODE_MTFTP_INFO;
```

MCastIp	File multicast IP address. This is the IP address to which the server will send the requested file.
CPort	Client multicast listening port. This is the UDP port to which the server will send the requested file.
SPort	Server multicast listening port. This is the UDP port on which the server listens for multicast open requests and data acks.

ListenTimeout	The number of seconds a client should listen for an active multicast session before requesting a new multicast session.
TransmitTimeout	The number of seconds a client should wait for a packet from the server before retransmitting the previous open request or data ack packet.

Description

This function is used to perform TFTP and MTFTP services. This includes the TFTP operations to get the size of a file, read a directory, read a file, and write a file. It also includes the MTFTP operations to get the size of a file, read a directory, and read a file. The type of operation is specified by *Operation*. If the callback function that is invoked during the TFTP/MTFTP operation does not return **EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE**, then **EFI_ABORTED** will be returned.

For read operations, the return data will be placed in the buffer specified by *BufferPtr*. If *BufferSize* is too small to contain the entire downloaded file, then **EFI_BUFFER_TOO_SMALL** will be returned and *BufferSize* will be set to zero or the size of the requested file (the size of the requested file is only returned if the TFTP server supports TFTP options). If *BufferSize* is large enough for the read operation, then *BufferSize* will be set to the size of the downloaded file, and **EFI_SUCCESS** will be returned. Applications using the **PxeBc.Mtftp()** services should use the get-file-size operations to determine the size of the downloaded file prior to using the read-file operations—especially when downloading large (greater than 64 MB) files—instead of making two calls to the read-file operation. Following this recommendation will save time if the file is larger than expected and the TFTP server does not support TFTP option extensions. Without TFTP option extension support, the client has to download the entire file, counting and discarding the received packets, to determine the file size.

For write operations, the data to be sent is in the buffer specified by *BufferPtr*. *BufferSize* specifies the number of bytes to send. If the write operation completes successfully, then **EFI_SUCCESS** will be returned.

For TFTP "get file size" operations, the size of the requested file or directory is returned in *BufferSize*, and **EFI_SUCCESS** will be returned. If the TFTP server does not support options, the file will be downloaded into a bit bucket and the length of the downloaded file will be returned. For MTFTP "get file size" operations, if the MTFTP server does not support the "get file size" option, **EFI_UNSUPPORTED** will be returned.

This function can take up to 10 seconds to timeout and return control to the caller. If the TFTP sequence does not complete, **EFI_TIMEOUT** will be returned.

If the Callback Protocol does not return **EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE**, then the TFTP sequence is stopped and **EFI_ABORTED** will be returned.

The format of the data returned from a TFTP read directory operation is a null-terminated filename followed by a null-terminated information string, of the form "size year-month-day hour:minute:second" (i.e. %d %d-%d-%d %d:%d:%f - note that the seconds field can be a decimal number), where the date and time are UTC. For an MTFTP read directory command, there is additionally a null-terminated multicast IP address preceding the filename of the form %d.%d.%d.%d for IP v4 (TBD for IP v6). The final entry is itself null-terminated, so that the final information string is terminated with two null octets.

Status Codes Returned

EFI_SUCCESS	The TFTP/MTFTP operation was completed.
EFI_NOT_STARTED	The PXE Base Code Protocol is in the stopped state.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_DEVICE_ERROR	The network device encountered an error during this operation.
EFI_BUFFER_TOO_SMALL	The buffer is not large enough to complete the read operation.
EFI_ABORTED	The callback function aborted the TFTP/MTFTP operation.
EFI_TIMEOUT	The TFTP/MTFTP operation timed out.
EFI_TFTP_ERROR	The TFTP/MTFTP operation generated an error.

EFI_PXE_BASE_CODE.UdpWrite()

Summary

Writes a UDP packet to the network interface.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_UDP_WRITE) (
     IN EFI_PXE_BASE_CODE                   *This,
     IN UINT16                              OpFlags,
     IN EFI_IP_ADDRESS                      *DestIp,
     IN EFI_PXE_BASE_CODE_UDP_PORT          *DestPort,
     IN EFI_IP_ADDRESS                      *GatewayIp,  OPTIONAL
     IN EFI_IP_ADDRESS                      *SrcIp,      OPTIONAL
     IN OUT EFI_PXE_BASE_CODE_UDP_PORT      *SrcPort,    OPTIONAL
     IN UINTN                               *HeaderSize, OPTIONAL
     IN VOID                                *HeaderPtr,  OPTIONAL
     IN UINTN                               *BufferSize,
     IN VOID                                *BufferPtr
     );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
OpFlags	The UDP operation flags. If **MAY_FRAGMENT** is set, then if required, this UDP write operation may be broken up across multiple packets.
DestIp	The destination IP address.
DestPort	The destination UDP port number.
GatewayIp	The gateway IP address. If *DestIp* is not in the same subnet as *StationIp*, then this gateway IP address will be used. If this field is **NULL**, and the *DestIp* is not in the same subnet as *StationIp*, then the *RouteTable* will be used.
SrcIp	The source IP address. If this field is *NULL*, then *StationIp* will be used as the source IP address.
SrcPort	The source UDP port number. If *OpFlags* has **ANY_SRC_PORT** set or *SrcPort* is **NULL**, then a source UDP port will be automatically selected. If a source UDP port was automatically selected, and *SrcPort* is not **NULL**, then it will be returned in *SrcPort*.
HeaderSize	An optional field which may be set to the length of a header at *HeaderPtr* to be prepended to the data at *BufferPtr*.

HeaderPtr	If *HeaderSize* is not **NULL**, a pointer to a header to be prepended to the data at *BufferPtr*.
BufferSize	A pointer to the size of the data at *BufferPtr*.
BufferPtr	A pointer to the data to be written.

Description

This function writes a UDP packet specified by the (optional *HeaderPtr* and) *BufferPtr* parameters to the network interface. The UDP header is automatically built by this routine. It uses the parameters *OpFlags*, *DestIp*, *DestPort*, *GatewayIp*, *SrcIp*, and *SrcPort* to build this header. If the packet is successfully built and transmitted through the network interface, then **EFI_SUCCESS** will be returned. If a timeout occurs during the transmission of the packet, then **EFI_TIMEOUT** will be returned. If an ICMP error occurs during the transmission of the packet, then the *IcmpErrorReceived* field is set to **TRUE**, the *IcmpError* field is filled in and **EFI_ICMP_ERROR** will be returned. If the Callback Protocol does not return **EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE**, then **EFI_ABORTED** will be returned.

Status Codes Returned

EFI_SUCCESS	The UDP Write operation was completed.
EFI_NOT_STARTED	The PXE Base Code Protocol is in the stopped state.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_DEVICE_ERROR	The network device encountered an error during this operation.
EFI_BAD_BUFFER_SIZE	The buffer is too long to be transmitted.
EFI_ABORTED	The callback function aborted the UDP Write operation.
EFI_TIMEOUT	The UDP Write operation timed out.
EFI_ICMP_ERROR	The UDP Write operation generated an error.

EFI_PXE_BASE_CODE.UdpRead()

Summary

Reads a UDP packet from the network interface.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_UDP_READ) (
      IN EFI_PXE_BASE_CODE                    *This
      IN UINT16                               OpFlags,
      IN OUT EFI_IP_ADDRESS                   *DestIp,      OPTIONAL
      IN OUT EFI_PXE_BASE_CODE_UDP_PORT       *DestPort,    OPTIONAL
      IN OUT EFI_IP_ADDRESS                   *SrcIp,       OPTIONAL
      IN OUT EFI_PXE_BASE_CODE_UDP_PORT       *SrcPort,     OPTIONAL
      IN UINTN                                *HeaderSize, OPTIONAL
      IN VOID                                 *HeaderPtr,   OPTIONAL
      IN OUT UINTN                            *BufferSize,
      IN VOID                                 *BufferPtr
      );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
OpFlags	The UDP operation flags.
DestIp	The destination IP address.
DestPort	The destination UDP port number.
SrcIp	The source IP address.
SrcPort	The source UDP port number.
HeaderSize	An optional field which may be set to the length of a header to be put in *HeaderPtr*.
HeaderPtr	If *HeaderSize* is not **NULL**, a pointer to a buffer to hold the *HeaderSize* bytes which follow the UDP header.
BufferSize	On input, a pointer to the size of the buffer at *BufferPtr*. On output, the size of the data written to *BufferPtr*.
BufferPtr	A pointer to the data to be read.

Description

This function reads a UDP packet from a network interface. The data contents are returned in (the optional *HeaderPtr* and) *BufferPtr*, and the size of the buffer received is returned in *BufferSize* . If the input *BufferSize* is smaller than the UDP packet received (less optional *HeaderSize*), it will be set to the required size, and **EFI_BUFFER_TOO_SMALL** will be returned. In this case, the contents of *BufferPtr* are undefined, and the packet is lost. If a UDP packet is successfully received, then **EFI_SUCCESS** will be returned, and the information from the UDP header will be returned in *DestIp*, *DestPort*, *SrcIp*, and *SrcPort* if they are not **NULL**. Depending on the values of *OpFlags* and the *DestIp*, *DestPort*, *SrcIp*, and *SrcPort* input values, different types of UDP packet receive filtering will be performed. The following tables summarize these receive filter operations.

Table 15-2. Destination IP Filter Operation

OpFlags USE_FILTER	OpFlags ANY_DEST_IP	DestIp	Action
0	0	NULL	Receive a packet sent to *StationIp*.
0	1	NULL	Receive a packet sent to any IP address.
1	x	NULL	Receive a packet whose destination IP address passes the IP filter.
0	0	not NULL	Receive a packet whose destination IP address matches *DestIp*.
0	1	not NULL	Receive a packet sent to any IP address and, return the destination IP address in *DestIp*.
1	x	not NULL	Receive a packet whose destination IP address passes the IP filter, and return the destination IP address in *DestIp*.

Table 15-3. Destination UDP Port Filter Operation

OpFlags ANY_DEST_PORT	DestPort	Action
0	NULL	Return **EFI_INVALID_PARAMETER**.
1	NULL	Receive a packet sent to any UDP port.
0	not NULL	Receive a packet whose destination Port matches *DestPort*.
1	not NULL	Receive a packet sent to any UDP port, and return the destination port in *DestPort*.

Table 15-4. Source IP Filter Operation

OpFlags ANY_SRC_IP	SrcIp	Action
0	NULL	Return **`EFI_INVALID_PARAMETER`**.
1	NULL	Receive a packet sent from any IP address.
0	not NULL	Receive a packet whose source IP address matches *`SrcIp`*.
1	not NULL	Receive a packet sent from any IP address, and return the source IP address in *`SrcIp`*.

Table 15-5. Source UDP Port Filter Operation

OpFlags ANY_SRC_PORT	SrcPort	Action
0	NULL	Return **`EFI_INVALID_PARAMETER`**.
1	NULL	Receive a packet sent from any UDP port.
0	not NULL	Receive a packet whose source UDP port matches *`SrcPort`*.
1	not NULL	Receive a packet sent from any UDP port, and return the source UPD port in *`SrcPort`*.

Status Codes Returned

EFI_SUCCESS	The UDP Read operation was completed.
EFI_NOT_STARTED	The PXE Base Code Protocol is in the stopped state.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_DEVICE_ERROR	The network device encountered an error during this operation.
EFI_BUFFER_TOO_SMALL	The packet is larger than *`Buffer`* can hold.
EFI_ABORTED	The callback function aborted the UDP Read operation.
EFI_TIMEOUT	The UDP Read operation timed out.
EFI_ICMP_ERROR	The UDP Read operation generated an error.

EFI_PXE_BASE_CODE.SetIpFilter()

Summary

Updates the IP receive filters of a network device and enables software filtering.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_SET_IP_FILTER) (
      IN EFI_PXE_BASE_CODE              *This,
      IN EFI_PXE_BASE_CODE_IP_FILTER    *NewFilter
      );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
NewFilter	Pointer to the new set of IP receive filters.

Description

The *NewFilter* field is used to modify the network device's current IP receive filter settings and to enable a software filter. This function updates the *IpFilter* field of the **EFI_PXE_BASE_CODE_MODE** structure with the contents of *NewIpFilter*. The software filter is used when the **USE_FILTER** in *OpFlags* is set to **UdpRead()**. The current hardware filter remains in effect no matter what the settings of *OpFlags* are, so that the meaning of **ANY_DEST_IP** set in *OpFlags* to **UdpRead()** is from those packets whose reception is enabled in hardware – physical NIC address (unicast), broadcast address, logical address or addresses (multicast), or all (promiscuous). **UdpRead()** does not modify the IP filter settings.

Dhcp(), **Discover()**, and **Mtftp()** set the IP filter, and return with the IP receive filter list emptied and the filter set to **EFI_PXE_BASE_CODE_IP_FILTER_STATION_IP**. If an application or driver wishes to preserve the IP receive filter settings, it will have to preserve the IP receive filter settings before these calls, and use **SetIpFilter()** to restore them after the calls. If incompatible filtering is requested (for example, **PROMISCUOUS** with anything else) or if the device does not support a requested filter setting and it cannot be accommodated in software (for example, **PROMISCUOUS** not supported), **EFI_INVALID_PARAMETER** will be returned. The *IPlist* field is used to enable IPs other than the *StationIP*. They may be multicast or unicast. If *IPcnt* is set as well as **EFI_PXE_BASE_CODE_IP_FILTER_STATION_IP**, then both the *StationIP* and the IPs from the *IPlist* will be used.

Status Codes Returned

EFI_SUCCESS	The IP receive filter settings were updated.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_NOT_STARTED	The PXE Base Code Protocol is not in the started state.

EFI_PXE_BASE_CODE.Arp()

Summary

Uses the ARP protocol to resolve a MAC address.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_ARP) (
      IN EFI_PXE_BASE_CODE                  *This,
      IN EFI_IP_ADDRESS                     *IpAddr,
      IN EFI_MAC_ADDRESS                    *MacAddr        OPTIONAL
      );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
IpAddr	Pointer to the IP address that is used to resolve a MAC address. When the MAC address is resolved, the *ArpCacheEntries* and *ArpCache* fields of the **EFI_PXE_BASE_CODE_MODE** structure are updated.
MacAddr	If not **NULL**, a pointer to the MAC address that was resolved with the ARP protocol.

Description

This function uses the ARP protocol to resolve a MAC address. The *UsingIpv6* field of the **EFI_PXE_BASE_CODE_MODE** structure is used to determine if IPv4 or IPv6 addresses are being used. The IP address specified by *IpAddr* is used to resolve a MAC address. If the ARP protocol succeeds in resolving the specified address, then the *ArpCacheEntries* and *ArpCache* fields of the **EFI_PXE_BASE_CODE_MODE** structure are updated, and **EFI_SUCCESS** is returned. If *MacAddr* is not **NULL**, the resolved MAC address is placed there as well.

If the PXE Base Code protocol is in the stopped state, then **EFI_NOT_STARTED** is returned. If the ARP protocol encounters a timeout condition while attempting to resolve an address, then **EFI_TIMEOUT** is returned. If the Callback Protocol does not return **EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE**, then **EFI_ABORTED** is returned.

Status Codes Returned

EFI_SUCCESS	The IP or MAC address was resolved.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_DEVICE_ERROR	The network device encountered an error during this operation.
EFI_NOT_STARTED	The PXE Base Code Protocol is in the stopped state.
EFI_TIMEOUT	The ARP Protocol encountered a timeout condition.
EFI_ABORTED	The callback function aborted the ARP Protocol.

EFI_PXE_BASE_CODE.SetParameters()

Summary

Updates the parameters that affect the operation of the PXE Base Code Protocol.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_SET_PARAMETERS) (
      IN EFI_PXE_BASE_CODE        *This,
      IN BOOLEAN                  *NewAutoArp,          OPTIONAL
      IN BOOLEAN                  *NewSendGUID,         OPTIONAL
      IN UINT8                    *NewTTL,              OPTIONAL
      IN UINT8                    *NewToS,              OPTIONAL
      IN BOOLEAN                  *NewMakeCallback      OPTIONAL
      );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
NewAutoArp	If not **NULL**, a pointer to a value that specifies whether to replace the current value of *AutoARP*. **TRUE** for automatic ARP packet generation, **FALSE** otherwise. If **NULL**, this parameter is ignored.
NewSendGUID	If not **NULL**, a pointer to a value that specifies whether to replace the current value of *SendGUID*. **TRUE** to send the SystemGUID (if there is one) as the client hardware address in DHCP; **FALSE** to send client NIC MAC address. If **NULL**, this parameter is ignored. If *NewSendGUID* is **TRUE** and there is no SystemGUID, then **EFI_INVALID_PARAMETER** is returned.
NewTTL	If not **NULL**, a pointer to be used in place of the current value of *TTL*, the “time to live” field of the IP header. If **NULL**, this parameter is ignored.
NewToS	If not **NULL**, a pointer to be used in place of the current value of *ToS*, the “type of service” field of the IP header. If **NULL**, this parameter is ignored.
NewMakeCallback	If not **NULL**, a pointer to a value that specifies whether to replace the current value of the *MakeCallback* field of the Mode structure. If **NULL**, this parameter is ignored. If the Callback Protocol is not available **EFI_INVALID_PARAMETER** is returned.

Description

This function sets parameters that affect the operation of the PXE Base Code Protocol. The parameter specified by *NewAutoArp* is used to control the generation of ARP protocol packets. If *NewAutoArp* is **TRUE**, then ARP Protocol packets will be generated as required by the PXE Base Code Protocol. If *NewAutoArp* is **FALSE**, then no ARP Protocol packets will be generated. In this case, the only mappings that are available are those stored in the *ArpCache* of the **EFI_PXE_BASE_CODE_MODE** structure. If there are not enough mappings in the *ArpCache* to perform a PXE Base Code Protocol service, then the service will fail. This function updates the *AutoArp* field of the **EFI_PXE_BASE_CODE_MODE** structure to *NewAutoArp*.

The **EFI_PXE_BASE_CODE.SetParameters()** call must be invoked after a Callback Protocol is installed to enable the use of callbacks.

Status Codes Returned

EFI_SUCCESS	The new parameters values were updated.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_NOT_STARTED	The PXE Base Code Protocol is not in the started state.

EFI_PXE_BASE_CODE.SetStationIp()

Summary

Updates the station IP address and/or subnet mask values of a network device.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_SET_STATION_IP) (
     IN EFI_PXE_BASE_CODE             *This,
     IN EFI_IP_ADDRESS                *NewStationIp,    OPTIONAL
     IN EFI_IP_ADDRESS                *NewSubnetMask    OPTIONAL
     );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
NewStationIp	Pointer to the new IP address to be used by the network device. If this field is **NULL**, then the *StationIp* address will not be modified.
NewSubnetMask	Pointer to the new subnet mask to be used by the network device. If this field is **NULL**, then the *SubnetMask* will not be modified.

Description

This function updates the station IP address and/or subnet mask values of a network device.

The *NewStationIp* field is used to modify the network device's current IP address. If *NewStationIP* is **NULL**, then the current IP address will not be modified. Otherwise, this function updates the *StationIp* field of the **EFI_PXE_BASE_CODE_MODE** structure with *NewStationIp*.

The *NewSubnetMask* field is used to modify the network device's current subnet mask. If *NewSubnetMask* is **NULL**, then the current subnet mask will not be modified. Otherwise, this function updates the *SubnetMask* field of the **EFI_PXE_BASE_CODE_MODE** structure with *NewSubnetMask*.

Status Codes Returned

EFI_SUCCESS	The new station IP address and/or subnet mask were updated.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_NOT_STARTED	The PXE Base Code Protocol is not in the started state.

EFI_PXE_BASE_CODE.SetPackets()

Summary

Updates the contents of the cached DHCP and Discover packets.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_SET_PACKETS) (
      IN EFI_PXE_BASE_CODE         *This,
      IN BOOLEAN                   *NewDhcpDiscoverValid,  OPTIONAL
      IN BOOLEAN                   *NewDhcpAckReceived,    OPTIONAL
      IN BOOLEAN                   *NewProxyOfferReceived, OPTIONAL
      IN BOOLEAN                   *NewPxeDiscoverValid,   OPTIONAL
      IN BOOLEAN                   *NewPxeReplyReceived,   OPTIONAL
      IN BOOLEAN                   *NewPxeBisReplyReceived,OPTIONAL
      IN EFI_PXE_BASE_CODE_PACKET       *NewDhcpDiscover, OPTIONAL
      IN EFI_PXE_BASE_CODE_PACKET       *NewDhcpAck,      OPTIONAL
      IN EFI_PXE_BASE_CODE_PACKET       *NewProxyOffer,   OPTIONAL
      IN EFI_PXE_BASE_CODE_PACKET       *NewPxeDiscover,  OPTIONAL
      IN EFI_PXE_BASE_CODE_PACKET       *NewPxeReply,     OPTIONAL
      IN EFI_PXE_BASE_CODE_PACKET       *NewPxeBisReply   OPTIONAL
      );
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
NewDhcpDiscoverValid	If not **NULL**, a pointer to a value that specifies whether to replace the current value of *DhcpDiscoverValid* field. If **NULL**, this parameter is ignored.
NewDhcpAckReceived	If not **NULL**, a pointer to a value that specifies whether to replace the current value of *DhcpAckReceived* field. If **NULL**, this parameter is ignored.
NewProxyOfferReceived	If not **NULL**, a pointer to a value that specifies whether to replace the current value of *ProxyOfferReceived* field. If **NULL**, this parameter is ignored.
NewPxeDiscoverValid	If not **NULL**, a pointer to a value that specifies whether to replace the current value of *PxeDiscoverValid* field. If **NULL**, this parameter is ignored.
NewPxeReplyReceived	If not **NULL**, a pointer to a value that specifies whether to replace the current value of *PxeReplyReceived* field. If **NULL**, this parameter is ignored.

NewPxeBisReplyReceived	If not **NULL**, a pointer to a value that specifies whether to replace the current value of *PxeBisReplyReceived* field. If **NULL**, this parameter is ignored.
NewDhcpDiscover	Pointer to the new cached DHCP Discover packet.
NewDhcpAck	Pointer to the new cached DHCP Ack packet.
NewProxyOffer	Pointer to the new cached Proxy Offer packet.
NewPxeDiscover	Pointer to the new cached PXE Discover packet.
NewPxeReply	Pointer to the new cached PXE Reply packet.
NewPxeBisReply	Pointer to the new cached PXE BIS Reply packet.

Description

The pointers to the new packets are used to update the contents of the cached packets in the **EFI_PXE_BASE_CODE_MODE** structure.

Status Codes Returned

EFI_SUCCESS	The cached packet contents were updated.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
EFI_NOT_STARTED	The PXE Base Code Protocol is not in the started state.

15.4 PXE Base Code Callback Protocol

This protocol is a specific instance of the PXE Base Code Callback Protocol that is invoked when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet. The PXE Base Code Callback Protocol must be on the same handle as the PXE Base Code Protocol.

EFI_PXE_BASE_CODE_CALLBACK Protocol

Summary

Protocol that is invoked when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet.

GUID

```
#define EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL \
      { 245DCA21-FB7B-11d3-8F01-00A0C969723B }
```

Revision Number

```
#define EFI_PXE_BASE_CODE_CALLBACK_INTERFACE_REVISION \
      0x00010000
```

Protocol Interface Structure

```
typedef struct {
      UINT64                                    Revision;
      EFI_PXE_CALLBACK                          Callback;
} EFI_PXE_BASE_CODE_CALLBACK;
```

Parameters

Revision	The revision of the **EFI_PXE_BASE_CODE_CALLBACK** protocol. All future revisions must be backwards compatible. If a future revision is not backwards compatible, it is not the same GUID.
Callback	Callback routine used by the PXE Base Code **Dhcp()**, **Discover()**, **Mtftp()**, **UdpWrite()**, and **Arp()** functions.

EFI_PXE_BASE_CODE_CALLBACK.Callback()

Summary

Callback function that is invoked when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet.

Prototype

```
EFI_PXE_BASE_CODE_CALLBACK_STATUS
(*EFI_PXE_CALLBACK) (
     IN EFI_PXE_BASE_CODE_CALLBACK          *This,
     IN EFI_PXE_BASE_CODE_FUNCTION          Function,
     IN BOOLEAN                             Received,
     IN UINT32                              PacketLen,
     IN EFI_PXE_BASE_CODE_PACKET            *Packet      OPTIONAL
);
```

Parameters

This	Pointer to the **EFI_PXE_BASE_CODE** instance.
Function	The PXE Base Code Protocol function that is waiting for an event.
Received	**TRUE** if the callback is being invoked due to a receive event. **FALSE** if the callback is being invoked due to a transmit event.
PacketLen	The length, in bytes, of *Packet*. This field will have a value of zero if this is a wait for receive event.
Packet	If *Received* is **TRUE**, a pointer to the packet that was just received; otherwise a pointer to the packet that is about to be transmitted. This field will be **NULL** if this is not a packet event.

Related Definitions

```
//*******************************************************
// EFI_PXE_BASE_CODE_CALLBACK_STATUS
//*******************************************************
typedef enum {
     EFI_PXE_BASE_CODE_CALLBACK_STATUS_FIRST,
     EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE,
     EFI_PXE_BASE_CODE_CALLBACK_STATUS_ABORT,
     EFI_PXE_BASE_CODE_CALLBACK_STATUS_LAST
} EFI_PXE_BASE_CODE_CALLBACK_STATUS;
```

```
//*******************************************************
// EFI_PXE_BASE_CODE_FUNCTION
//*******************************************************
typedef enum {
     EFI_PXE_BASE_CODE_FUNCTION_FIRST,
     EFI_PXE_BASE_CODE_FUNCTION_DHCP,
     EFI_PXE_BASE_CODE_FUNCTION_DISCOVER,
     EFI_PXE_BASE_CODE_FUNCTION_MTFTP,
     EFI_PXE_BASE_CODE_FUNCTION_UDP_WRITE,
     EFI_PXE_BASE_CODE_FUNCTION_UDP_READ,
     EFI_PXE_BASE_CODE_FUNCTION_ARP,
     EFI_PXE_BASE_CODE_FUNCTION_IGMP,
     EFI_PXE_BASE_CODE_PXE_FUNCTION_LAST
} EFI_PXE_BASE_CODE_FUNCTION;
```

Description

This function is invoked when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet. Parameters *Function* and *Received* specify the type of event. Parameters *PacketLen* and *Packet* specify the packet that generated the event. If these fields are zero and **NULL** respectively, then this is a status update callback. If the operation specified by *Function* is to continue, then **CALLBACK_STATUS_CONTINUE** should be returned. If the operation specified by *Function* should be aborted, then **CALLBACK_STATUS_ABORT** should be returned. Due to the polling nature of EFI device drivers, a callback function should not execute for more than 5 ms.

The **EFI_PXE_BASE_CODE.SetParameters()** function must be called after a Callback Protocol is installed to enable the use of callbacks.

15.5 Boot Integrity Services Protocol

This chapter defines the Boot Integrity Services (BIS) protocol, which is used to check a digital signature of a data block against a digital certificate for the purpose of an integrity and authorization check. BIS is primarily used by the Preboot Execution Environment (PXE) Base Code protocol **EFI_PXE_BASE_CODE_PROTOCOL** to check downloaded network boot images before executing them. BIS is an EFI Boot Services Driver, so its services are also available to EFI Applications until the time of **ExitBootServices()**. More information about BIS can be found in the *Boot Integrity Services Application Programming Interface Version 1.0.*

This section defines the Boot Integrity Services Protocol. This protocol is used to check a digital signature of a data block against a digital certificate for the purpose of an integrity and authorization check.

EFI_BIS_PROTOCOL

Summary

The **EFI_BIS_PROTOCOL** is used to check a digital signature of a data block against a digital certificate for the purpose of an integrity and authorization check.

GUID

```
#define EFI_BIS_PROTOCOL_GUID        \
 {0x0b64aab0,0x5429,0x11d4,0x98,0x16,0x00,0xa0,0xc9,0x1f,0xad,0xcf}
```

Protocol Interface Structure

```
typedef struct _EFI_BIS_PROTOCOL {
    EFI_BIS_INITIALIZE      Initialize;
    EFI_BIS_SHUTDOWN        Shutdown;
    EFI_BIS_FREE            Free;
    EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_CERTIFICATE
                            GetBootObjectAuthorizationCertificate;
    EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_CHECKFLAG
                            GetBootObjectAuthorizationCheckFlag;
    EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_UPDATE_TOKEN
                            GetBootObjectAuthorizationUpdateToken;
    EFI_BIS_GET_SIGNATURE_INFO
                            GetSignatureInfo;
    EFI_BIS_UPDATE_BOOT_OBJECT_AUTHORIZATION
                            UpdateBootObjectAuthorization;
    EFI_BIS_VERIFY_BOOT_OBJECT
                            VerifyBootObject;
    EFI_BIS_VERIFY_OBJECT_WITH_CREDENTIAL
                            VerifyObjectWithCredential;
} EFI_BIS_PROTOCOL;
```

Parameters

Initialize	Initializes an application instance of the **EFI_BIS** protocol, returning a handle for the application instance. Other functions in the **EFI_BIS** protocol require a valid application instance handle obtained from this function. See the **Initialize()** function description.
Shutdown	Ends the lifetime of an application instance of the **EFI_BIS** protocol, invalidating its application instance handle. The application instance handle may no longer be used in other functions in the **EFI_BIS** protocol. See the **Shutdown()** function description.
Free	Frees memory structures allocated and returned by other functions in the **EFI_BIS** protocol. See the **Free()** function description.
GetBootObjectAuthorizationCertificate	Retrieves the current digital certificate (if any) used by the **EFI_BIS** protocol as the source of authorization for verifying boot objects and altering configuration parameters. See the **GetBootObjectAuthorizationCertificate()** function description.
GetBootObjectAuthorizationCheckFlag	Retrieves the current setting of the authorization check flag that indicates whether or not authorization checks are required for boot objects. See the **GetBootObjectAuthorizationCheckFlag()** function description.
GetBootObjectAuthorizationUpdateToken	Retrieves an uninterpreted token whose value gets included and signed in a subsequent request to alter the configuration parameters, to protect against attempts to "replay" such a request. See the **GetBootObjectAuthorizationUpdateToken()** function description.
GetSignatureInfo	Retrieves information about the digital signature algorithms supported and the identity of the installed authorization certificate, if any. See the **GetSignatureInfo()** function description.
UpdateBootObjectAuthorization	Requests that the configuration parameters be altered by installing or removing an authorization certificate or changing the setting of the check flag. See the **UpdateBootObjectAuthorization()** function description.

`VerifyBootObject`

Verifies a boot object according to the supplied digital signature and the current authorization certificate and check flag setting. See the **`VerifyBootObject()`** function description.

`VerifyObjectWithCredential`

Verifies a data object according to a supplied digital signature and a supplied digital certificate. See the **`VerifyObjectWithCredential()`** function description.

Description

The **`EFI_BIS_PROTOCOL`** provides a set of functions as defined in this chapter. There is no physical device associated with these functions, however, in the context of EFI every protocol operates on a device. Accordingly, BIS installs and operates on a single abstract device that has only a software representation.

EFI_BIS.Initialize()

Summary

Initializes the BIS service, checking that it is compatible with the version requested by the caller. After this call, other BIS functions may be invoked.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_INITIALIZE)(
  IN      EFI_BIS_PROTOCOL          *This,
  OUT     BIS_APPLICATION_HANDLE    *AppHandle,
  IN OUT  EFI_BIS_VERSION           *InterfaceVersion,
  IN      EFI_BIS_DATA              *TargetAddress
  );
```

Parameters

This	A pointer to the **EFI_BIS_PROTOCOL** object. The protocol implementation may rely on the actual pointer value and object location, so the caller must not copy the object to a new location.
AppHandle	The function writes the new **BIS_APPLICATION_HANDLE** if successful, otherwise it writes **NULL**. The caller must eventually destroy this handle by calling **Shutdown()**. Type **BIS_APPLICATION_HANDLE** is defined in "Related Definitions" below.
InterfaceVersion	On input, the caller supplies the major version number of the interface version desired. The minor version number supplied on input is ignored since interface compatibility is determined solely by the major version number. On output, both the major and minor version numbers are updated with the major and minor version numbers of the interface (and underlying implementation). This update is done whether or not the initialization was successful. Type **EFI_BIS_VERSION** is defined in "Related Definitions" below.
TargetAddress	Indicates a network or device address of the BIS platform to connect to. Local-platform BIS implementations require that the caller sets *TargetAddress.Data* to **NULL**, but otherwise ignores this parameter. BIS implementations that redirect calls to an agent at a remote address must define their own format and interpretation of this parameter outside the scope of this document. For all implementations, if the *TargetAddress* is an unsupported value, the function fails with the error **EFI_UNSUPPORTED**. Type **EFI_BIS_DATA** is defined in "Related Definitions" below.

Related Definitions

```
//*******************************************************
// BIS_APPLICATION_HANDLE
//*******************************************************
typedef VOID                   *BIS_APPLICATION_HANDLE;
```

This type is an opaque handle representing an initialized instance of the BIS interface. A **BIS_APPLICATION_HANDLE** value is returned by the **Initialize()** function as an "out" parameter. Other BIS functions take a **BIS_APPLICATION_HANDLE** as an "in" parameter to identify the BIS instance.

```
//*******************************************************
// EFI_BIS_VERSION
//*******************************************************
typedef struct _EFI_BIS_VERSION {
  UINT32                Major;
  UINT32                Minor;
} EFI_BIS_VERSION;
```

Major	This describes the major BIS version number. The major version number defines version compatibility. That is, when a new version of the BIS interface is created with new capabilities that are not available in the previous interface version, the major version number is increased.
Minor	This describes a minor BIS version number. This version number is increased whenever a new BIS implementation is built that is fully interface compatible with the previous BIS implementation. This number may be reset when the major version number is increased.

This type represents a version number of the BIS interface. This is used as an "in out" parameter of the **Initialize()** function for a simple form of negotiation of the BIS interface version between the caller and the BIS implementation.

```
//*********************************************************
// EFI_BIS_VERSION predefined values
//   Use these values to initialize EFI_BIS_VERSION.Major
//   and to interpret results of Initialize.
//*********************************************************
#define BIS_CURRENT_VERSION_MAJOR      BIS_VERSION_1
#define BIS_VERSION_1                  1
```

These C preprocessor macros supply values for the major version number of an **EFI_BIS_VERSION**. At the time of initialization, a caller supplies a value to request a BIS interface version. On return, the (IN OUT) parameter is over-written with the actual version of the interface.

```
//*********************************************************
// EFI_BIS_DATA
//
// EFI_BIS_DATA instances obtained from BIS must be freed by
// calling Free().
//*********************************************************
typedef struct _EFI_BIS_DATA {
  UINT32        Length;
  UINT8         *Data;
} EFI_BIS_DATA;
```

Length	The length of the data buffer in bytes.
Data	A pointer to the raw data buffer.

This type defines a structure that describes a buffer. BIS uses this type to pass back and forth most large objects such as digital certificates, strings, etc.. Several of the BIS functions allocate a **EFI_BIS_DATA*** and return it as an "out" parameter. The caller must eventually free any allocated **EFI_BIS_DATA*** using the **Free()** function.

Description

This function must be the first BIS function invoked by an application. It passes back a **BIS_APPLICATION_HANDLE** value that must be used in subsequent BIS functions. The handle must be eventually destroyed by a call to the **Shutdown()** function, thus ending that handle's lifetime. After the handle is destroyed, BIS functions may no longer be called with that handle value. Thus all other BIS functions may only be called between a pair of **Initialize()** and **Shutdown()** functions.

There is no penalty for calling **Initialize()** multiple times. Each call passes back a distinct handle value. Each distinct handle must be destroyed by a distinct call to **Shutdown()**. The lifetimes of handles created and destroyed with these functions may be overlapped in any way.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_INCOMPATIBLE_VERSION	The *InterfaceVersion.Major* requested by the caller was not compatible with the interface version of the implementation. The *InterfaceVersion.Major* has been updated with the current interface version.
EFI_UNSUPPORTED	This is a local-platform implementation and *TargetAddress.Data* was not **NULL**, or *TargetAddress.Data* was any other value that was not supported by the implementation.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.
EFI_DEVICE_ERROR	The function encountered an unexpected internal failure while initializing a cryptographic software module, or No cryptographic software module with compatible version was found, or A resource limitation was encountered while using a cryptographic software module.
EFI_INVALID_PARAMETER	The *This* parameter supplied by the caller is **NULL** or does not reference a valid **EFI_BIS_PROTOCOL** object, or The *AppHandle* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *InterfaceVersion* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *TargetAddress* parameter supplied by the caller is **NULL** or an invalid memory reference.

EFI_BIS.Shutdown()

Summary

Shuts down an application's instance of the BIS service, invalidating the application handle. After this call, other BIS functions may no longer be invoked using the application handle value.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_SHUTDOWN)(
  IN BIS_APPLICATION_HANDLE  AppHandle
  );
```

Parameters

AppHandle	An opaque handle that identifies the caller's instance of initialization of the BIS service. Type **BIS_APPLICATION_HANDLE** is defined in the **Initialize()** function description.

Description

This function shuts down an application's instance of the BIS service, invalidating the application handle. After this call, other BIS functions may no longer be invoked using the application handle value.

This function must be paired with a preceding successful call to the **Initialize()** function. The lifetime of an application handle extends from the time the handle was returned from **Initialize()** until the time the handle is passed to **Shutdown()**. If there are other remaining handles whose lifetime is still active, they may still be used in calling BIS functions.

The caller must free all memory resources associated with this *AppHandle* that were allocated and returned from other BIS functions before calling **Shutdown()**. Memory resources are freed using the **Free()** function. Failure to free such memory resources is a caller error, however, this function does not return an error code under this circumstance. Further attempts to access the outstanding memory resources causes unspecified results.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NO_MAPPING	The *AppHandle* parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.
EFI_DEVICE_ERROR	The function encountered an unexpected internal error while returning resources associated with a cryptographic software module, or The function encountered an internal error while trying to shut down a cryptographic software module.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.

EFI_BIS.Free()

Summary

Frees memory structures allocated and returned by other functions in the **EFI_BIS** protocol.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_FREE)(
  IN BIS_APPLICATION_HANDLE  AppHandle,
  IN EFI_BIS_DATA            *ToFree
  );
```

Parameters

AppHandle	An opaque handle that identifies the caller's instance of initialization of the BIS service. Type **BIS_APPLICATION_HANDLE** is defined in the **Initialize()** function description.
ToFree	An **EFI_BIS_DATA*** and associated memory block to be freed. This **EFI_BIS_DATA*** must have been allocated by one of the other BIS functions. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.

Description

This function deallocates an **EFI_BIS_DATA*** and associated memory allocated by one of the other BIS functions.

Callers of other BIS functions that allocate memory in the form of an **EFI_BIS_DATA*** must eventually call this function to deallocate the memory before calling the **Shutdown()** function for the application handle under which the memory was allocated. Failure to do so causes unspecified results, and the continued correct operation of the BIS service cannot be guaranteed.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NO_MAPPING	The *AppHandle* parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.
EFI_INVALID_PARAMETER	The *ToFree* parameter is not or is no longer a memory resource associated with this *AppHandle*.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.

EFI_BIS.GetBootObjectAuthorizationCertificate()

Summary

Retrieves the certificate that has been configured as the identity of the organization designated as the source of authorization for signatures of boot objects.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_CERTIFICATE)(
  IN  BIS_APPLICATION_HANDLE   AppHandle,
  OUT EFI_BIS_DATA             **Certificate
  );
```

Parameters

AppHandle	An opaque handle that identifies the caller's instance of initialization of the BIS service. Type **BIS_APPLICATION_HANDLE** is defined in the **Initialize()** function description.
Certificate	The function writes an allocated **EFI_BIS_DATA*** containing the Boot Object Authorization Certificate object. The caller must eventually free the memory allocated by this function using the function **Free()**. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.

Description

This function retrieves the certificate that has been configured as the identity of the organization designated as the source of authorization for signatures of boot objects.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NO_MAPPING	The *AppHandle* parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.
EFI_NOT_FOUND	There is no Boot Object Authorization Certificate currently installed.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.
EFI_INVALID_PARAMETER	The *Certificate* parameter supplied by the caller is **NULL** or an invalid memory reference.

EFI_BIS.GetBootObjectAuthorizationCheckFlag()

Summary

Retrieves the current status of the Boot Authorization Check Flag.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_CHECKFLAG)(
  IN  BIS_APPLICATION_HANDLE  AppHandle,
  OUT BOOLEAN                 *CheckIsRequired
  );
```

Parameters

AppHandle	An opaque handle that identifies the caller's instance of initialization of the BIS service. Type **BIS_APPLICATION_HANDLE** is defined in the **Initialize()** function description.
CheckIsRequired	The function writes the value **TRUE** if a Boot Authorization Check is currently required on this platform, otherwise the function writes **FALSE**.

Description

This function retrieves the current status of the Boot Authorization Check Flag (in other words, whether or not a Boot Authorization Check is currently required on this platform).

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NO_MAPPING	The *AppHandle* parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.
EFI_INVALID_PARAMETER	The *CheckIsRequired* parameter supplied by the caller is **NULL** or an invalid memory reference.

EFI_BIS.GetBootObjectAuthorizationUpdateToken()

Summary

Retrieves a unique token value to be included in the request credential for the next update of any parameter in the Boot Object Authorization set (Boot Object Authorization Certificate and Boot Authorization Check Flag).

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_UPDATE_TOKEN)(
  IN  BIS_APPLICATION_HANDLE  AppHandle,
  OUT EFI_BIS_DATA            **UpdateToken
  );
```

Parameters

AppHandle	An opaque handle that identifies the caller's instance of initialization of the BIS service. Type **BIS_APPLICATION_HANDLE** is defined in the **Initialize()** function description.
UpdateToken	The function writes an allocated **EFI_BIS_DATA*** containing the new unique update token value. The caller must eventually free the memory allocated by this function using the function **Free()**. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.

Description

This function retrieves a unique token value to be included in the request credential for the next update of any parameter in the Boot Object Authorization set (Boot Object Authorization Certificate and Boot Authorization Check Flag). The token value is unique to this platform, parameter set, and instance of parameter values. In particular, the token changes to a new unique value whenever any parameter in this set is changed.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NO_MAPPING	The *AppHandle* parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.
EFI_DEVICE_ERROR	The function encountered an unexpected internal error in a cryptographic software module.
EFI_INVALID_PARAMETER	The *UpdateToken* parameter supplied by the caller is **NULL** or an invalid memory reference.

EFI_BIS.GetSignatureInfo()

Summary

Retrieves a list of digital certificate identifier, digital signature algorithm, hash algorithm, and key-length combinations that the platform supports.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_GET_SIGNATURE_INFO)(
  IN  BIS_APPLICATION_HANDLE   AppHandle,
  OUT EFI_BIS_DATA             **SignatureInfo
  );
```

Parameters

AppHandle	An opaque handle that identifies the caller's instance of initialization of the BIS service. Type **BIS_APPLICATION_HANDLE** is defined in the **Initialize()** function description.
SignatureInfo	The function writes an allocated **EFI_BIS_DATA*** containing the array of **EFI_BIS_SIGNATURE_INFO** structures representing the supported digital certificate identifier, algorithm, and key length combinations. The caller must eventually free the memory allocated by this function using the function **Free()**. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description. Type **EFI_BIS_SIGNATURE_INFO** is defined in "Related Definitions" below.

Related Definitions

```
//*********************************************************
// EFI_BIS_SIGNATURE_INFO
//*********************************************************
typedef struct _EFI_BIS_SIGNATURE_INFO {
  BIS_CERT_ID     CertificateID;
  BIS_ALG_ID      AlgorithmID;
  UINT16          KeyLength;
} EFI_BIS_SIGNATURE_INFO;
```

CertificateID	A shortened value identifying the platform's currently configured Boot Object Authorization Certificate, if one is currently configured. The shortened value is derived from the certificate as defined in the Related Definition for **BIS_CERT_ID** below. If there is no certificate currently configured, the value is one of the reserved **BIS_CERT_ID_XXX** values defined below. Type **BIS_CERT_ID** and its predefined reserved values are defined in "Related Definitions" below.

AlgorithmID	A predefined constant representing a particular digital signature algorithm. Often this represents a combination of hash algorithm and encryption algorithm, however, it may also represent a standalone digital signature algorithm. Type **BIS_ALG_ID** and its permitted values are defined in "Related Definitions" below.
KeyLength	The length of the public key, in bits, supported by this digital signature algorithm.

This type defines a digital certificate, digital signature algorithm, and key-length combination that may be supported by the BIS implementation. This type is returned by **GetSignatureInfo()** to describe the combination(s) supported by the implementation.

```
//*********************************************************
// BIS_GET_SIGINFO_COUNT macro
//   Tells how many EFI_BIS_SIGNATURE_INFO elements are contained
//   in a EFI_BIS_DATA struct pointed to by the provided
//   EFI_BIS_DATA*.
//*********************************************************
#define BIS_GET_SIGINFO_COUNT(BisDataPtr) \
  ((BisDataPtr)->Length/sizeof(EFI_BIS_SIGNATURE_INFO))
```

BisDataPtr	Supplies the pointer to the target **EFI_BIS_DATA** structure.
(return value)	The number of **EFI_BIS_SIGNATURE_INFO** elements contained in the array.

This macro computes how many **EFI_BIS_SIGNATURE_INFO** elements are contained in an **EFI_BIS_DATA** structure returned from **GetSignatureInfo()**. The number returned is the count of items in the list of supported digital certificate, digital signature algorithm, and key-length combinations.

```
//*********************************************************
// BIS_GET_SIGINFO_ARRAY macro
//   Produces a EFI_BIS_SIGNATURE_INFO* from a given
//   EFI_BIS_DATA*.
//*********************************************************
#define BIS_GET_SIGINFO_ARRAY(BisDataPtr) \
  ((EFI_BIS_SIGNATURE_INFO*)(BisDataPtr)->Data)
```

BisDataPtr	Supplies the pointer to the target **EFI_BIS_DATA** structure.
(return value)	The pointer to the **EFI_BIS_SIGNATURE_INFO** array, cast as an **EFI_BIS_SIGNATURE_INFO***.

This macro returns a pointer to the **EFI_BIS_SIGNATURE_INFO** array contained in an **EFI_BIS_DATA** structure returned from **GetSignatureInfo()** representing the list of supported digital certificate, digital signature algorithm, and key-length combinations.

```
//*******************************************************
// BIS_CERT_ID
//*******************************************************
typedef UINT32                  BIS_CERT_ID;
```

This type represents a shortened value that identifies the platform's currently configured Boot Object Authorization Certificate. The value is the first four bytes, in "little-endian" order, of the SHA-1 hash of the certificate, except that the most-significant bits of the second and third bytes are reserved, and must be set to zero regardless of the outcome of the hash function. This type is included in the array of values returned from the `GetSignatureInfo()` function to indicate the required source of a signature for a boot object or a configuration update request. There are a few predefined reserved values with special meanings as described below.

```
//*******************************************************
// BIS_CERT_ID predefined values
//   Currently defined values for EFI_BIS_SIGNATURE_INFO.
//   CertificateId.
//*******************************************************
#define BIS_CERT_ID_DSA     BIS_ALG_DSA      //CSSM_ALGID_DSA
#define BIS_CERT_ID_RSA_MD5 BIS_ALG_RSA_MD5  //CSSM_ALGID_MD5_WITH_RSA
```

These C preprocessor symbols provide values for the `BIS_CERT_ID` type. These values are used when the platform has no configured Boot Object Authorization Certificate. They indicate the signature algorithm that is supported by the platform. Users must be careful to avoid constructing Boot Object Authorization Certificates that transform to `BIS_CERT_ID` values that collide with these predefined values or with the `BIS_CERT_ID` values of other Boot Object Authorization Certificates they use.

```
//*******************************************************
// BIS_CERT_ID_MASK
//   The  following  is a mask value that gets applied to the
//   truncated hash of a platform  Boot Object Authorization
//   Certificate to create the CertificateId.  A CertificateId
//   must not have any bits set to the value 1 other than bits in
//   this mask.
//*******************************************************
#define BIS_CERT_ID_MASK (0xFF7F7FFF)
```

This C preprocessor symbol may be used as a bit-wise "AND" value to transform the first four bytes (in little-endian order) of a SHA-1 hash of a certificate into a certificate ID with the "reserved" bits properly set to zero.

```
//*********************************************************
// BIS_ALG_ID
//*********************************************************
typedef UINT16                  BIS_ALG_ID;
```

This type represents a digital signature algorithm. A digital signature algorithm is often composed of a particular combination of secure hash algorithm and encryption algorithm. This type also allows for digital signature algorithms that cannot be decomposed. Predefined values for this type are as defined below.

```
//*********************************************************
// BIS_ALG_ID predefined values
//   Currently defined values for EFI_BIS_SIGNATURE_INFO.
//   AlgorithmID.  The exact numeric values come from "Common
//   Data Security Architecture (CDSA) Specification."
//*********************************************************
#define BIS_ALG_DSA     (41)    //CSSM_ALGID_DSA
#define BIS_ALG_RSA_MD5 (42)    //CSSM_ALGID_MD5_WITH_RSA
```

These values represent the two digital signature algorithms predefined for BIS. Each implementation of BIS must support at least one of these digital signature algorithms. Values for the digital signature algorithms are chosen by an industry group known as The Open Group. Developers planning to support additional digital signature algorithms or define new digital signature algorithms should refer to The Open Group for interoperable values to use.

Description

This function retrieves a list of digital certificate identifier, digital signature algorithm, hash algorithm, and key-length combinations that the platform supports. The list is an array of (certificate id, algorithm id, key length) triples, where the certificate id is derived from the platform's Boot Object Authorization Certificate as described in the Related Definition for **BIS_CERT_ID** above, the algorithm id represents the combination of signature algorithm and hash algorithm, and the key length is expressed in bits. The number of array elements can be computed using the *Length* field of the retrieved **EFI_BIS_DATA***.

The retrieved list is in order of preference. A digital signature algorithm for which the platform has a currently configured Boot Object Authorization Certificate is preferred over any digital signature algorithm for which there is not a currently configured Boot Object Authorization Certificate. Thus the first element in the list has a *CertificateID* representing a Boot Object Authorization Certificate if the platform has one configured. Otherwise the *CertificateID* of the first element in the list is one of the reserved values representing a digital signature algorithm.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NO_MAPPING	The *AppHandle* parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.
EFI_DEVICE_ERROR	The function encountered an unexpected internal error in a cryptographic software module, or The function encountered an unexpected internal consistency check failure (possible corruption of stored Boot Object Authorization Certificate).
EFI_INVALID_PARAMETER	The *SignatureInfo* parameter supplied by the caller is **NULL** or an invalid memory reference.

EFI_BIS.UpdateBootObjectAuthorization()

Summary

Updates one of the configurable parameters of the Boot Object Authorization set (Boot Object Authorization Certificate or Boot Authorization Check Flag).

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_UPDATE_BOOT_OBJECT_AUTHORIZATION)(
    IN  BIS_APPLICATION_HANDLE AppHandle,
    IN  EFI_BIS_DATA           *RequestCredential,
    OUT EFI_BIS_DATA           **NewUpdateToken
    );
```

Parameters

AppHandle An opaque handle that identifies the caller's instance of initialization of the BIS service. Type **BIS_APPLICATION_HANDLE** is defined in the **Initialize()** function description.

RequestCredential

This is a Signed Manifest with embedded attributes that carry the details of the requested update. The required syntax of the Signed Manifest is described in the Related Definition for Manifest Syntax below. The key used to sign the request credential must be the private key corresponding to the public key in the platform's configured Boot Object Authorization Certificate. Authority to update parameters in the Boot Object Authorization set cannot be delegated.

If there is no Boot Object Authorization Certificate, the request credential may be signed with any private key. In this case, this function interacts with the user in a platform-specific way to determine whether the operation should succeed. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.

NewUpdateToken The function writes an allocated **EFI_BIS_DATA*** containing the new unique update token value. The caller must eventually free the memory allocated by this function using the function **Free()**. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.

Related Definitions

```
//*******************************************************
// Manifest Syntax
//*******************************************************
```

The Signed Manifest consists of three parts grouped together into an Electronic Shrink Wrap archive as described in [SM spec]: a manifest file, a signer's information file, and a signature block file. These three parts, along with examples are described in the following sections. In these examples, text in parentheses is a description of the text that would appear in the signed manifest. Text outside of parentheses must appear exactly as shown. Also note that manifest files and signer's information files must conform to a 72-byte line-length limit. Continuation lines (lines beginning with a single "space" character) are used for lines longer than 72 bytes. The examples given here follow this rule for continuation lines.

Note that the manifest file and signer's information file parts of a Signed Manifest are ASCII (not Unicode) text files. In cases where these files contain a base-64 encoded string, the string is an ASCII (not Unicode) string before base-64 encoding.

```
//*******************************************************
// Manifest File Example
//*******************************************************
```

The manifest file must include a section referring to a memory-type data object with the reserved name as shown in the example below. This data object is a zero-length object whose sole purpose in the manifest is to serve as a named collection point for the attributes that carry the details of the requested update. The attributes are also contained in the manifest file. An example manifest file is shown below.

```
Manifest-Version: 2.0
ManifestPersistentId: (base-64 representation of a unique GUID)

Name: memory:UpdateRequestParameters
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of zero-length
 buffer)
X-Intel-BIS-ParameterSet: (base-64 representation of
 BootObjectAuthorizationSetGUID)
X-Intel-BIS-ParameterSetToken: (base-64 representation of the current
 update token)
X-Intel-BIS-ParameterId: (base-64 representation of
 "BootObjectAuthorizationCertificate" or
 "BootAuthorizationCheckFlag")
X-Intel-BIS-ParameterValue: (base-64 representation of
 certificate or
 single-byte boolean flag)
```

A line-by-line description of this manifest file is as follows.

```
Manifest-Version: 2.0
```

This is a standard header line that all signed manifests have. It must appear exactly as shown.

```
ManifestPersistentId: (base-64 representation of a unique GUID)
```

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every manifest file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

```
Name: memory:UpdateRequestParameters
```

This identifies the manifest section that carries a dummy zero-length data object serving as the collection point for the attribute values appearing later in this manifest section (lines prefixed with "**`X-Intel-BIS-`**"). The string "**`memory:UpdateRequestParameters`**" must appear exactly as shown.

```
Digest-Algorithms: SHA-1
```

This enumerates the digest algorithms for which integrity data is included for the data object. These are required even though the data object is zero-length. For systems with DSA signing, SHA-1 hash, and 1024-bit key length, the digest algorithm must be "**`SHA-1`**." For systems with RSA signing, MD5 hash, and 512-bit key length, the digest algorithm must be "**`MD5`**." Multiple algorithms can be specified as a whitespace-separated list. For every digest algorithm **`XXX`** listed, there must also be a corresponding **`XXX-Digest`** line.

```
SHA-1-Digest: (base-64 representation of a SHA-1 digest of zero-length
 buffer)
```

Gives the corresponding digest value for the dummy zero-length data object. The value is base-64 encoded. Note that for both MD5 and SHA-1, the digest value for a zero-length data object is not zero.

```
X-Intel-BIS-ParameterSet: (base-64 representation of
 BootObjectAuthorizationSetGUID)
```

A named attribute value that distinguishes updates of BIS parameters from updates of other parameters. The left-hand attribute-name keyword must appear exactly as shown. The GUID value for the right-hand side is always the same, and can be found under the preprocessor symbol **`BOOT_OBJECT_AUTHORIZATION_PARMSET_GUIDVALUE`**. The representation inserted into the manifest is base-64 encoded.

Note the "**`X-Intel-BIS-`**" prefix on this and the following attributes. The "**`X-`**" part of the prefix was chosen to avoid collisions with future reserved keywords defined by future versions of the signed manifest specification. The "**`Intel-BIS-`**" part of the prefix was chosen to avoid collisions with other user-defined attribute names within the user-defined attribute name space.

```
X-Intel-BIS-ParameterSetToken: (base-64 representation of the current
 update token)
```

A named attribute value that makes this update of BIS parameters different from any other on the same target platform. The left-hand attribute-name keyword must appear exactly as shown. The value for the right-hand side is generally different for each update-request manifest generated. The value to be base-64 encoded is retrieved through the functions **`GetBootObjectAuthorizationUpdateToken()`** or **`UpdateBootObjectAuthorization()`**.

```
X-Intel-BIS-ParameterId: (base-64 representation of
 "BootObjectAuthorizationCertificate" or
 "BootAuthorizationCheckFlag")
```

A named attribute value that indicates which BIS parameter is to be updated. The left-hand attribute-name keyword must appear exactly as shown. The value for the right-hand side is the base-64 encoded representation of one of the two strings shown.

```
X-Intel-BIS-ParameterValue: (base-64 representation of
 certificate or
 single-byte boolean flag)
```

A named attribute value that indicates the new value to be set for the indicated parameter. The left-hand attribute-name keyword must appear exactly as shown. The value for the right-hand side is the appropriate base-64 encoded new value to be set. In the case of the Boot Object Authorization Certificate, the value is the new digital certificate raw data. A zero-length value removes the certificate altogether. In the case of the Boot Authorization Check Flag, the value is a single-byte boolean value, where a nonzero value "turns on" the check and a zero value "turns off" the check.

```
//************************************************************
// Signer's Information File Example
//************************************************************
```

The signer's information file must include a section whose name matches the reserved data object section name of the section in the Manifest file. This section in the signer's information file carries the integrity data for the attributes in the corresponding section in the manifest file. An example signer's information file is shown below.

```
Signature-Version: 2.0
SignerInformationPersistentId: (base-64 representation of a unique
 GUID)
SignerInformationName: BIS_UpdateManifestSignerInfoName

Name: memory:UpdateRequestParameters
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the
 corresponding manifest section)
```

A line-by-line description of this signer's information file is as follows.

```
Signature-Version: 2.0
```

This is a standard header line that all signed manifests have. It must appear exactly as shown.

```
SignerInformationPersistentId: (base-64 representation of a unique
 GUID)
```

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every signer's information file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

```
SignerInformationName: BIS_UpdateManifestSignerInfoName
```

The left-hand string must appear exactly as shown. The right-hand string must appear exactly as shown.

```
Name: memory:UpdateRequestParameters
```

This identifies the section in the signer's information file corresponding to the section with the same name in the manifest file described earlier. The string "**`memory:UpdateRequestParameters`**" must appear exactly as shown.

```
Digest-Algorithms: SHA-1
```

This enumerates the digest algorithms for which integrity data is included for the corresponding manifest section. Strings identifying digest algorithms are the same as in the manifest file. The digest algorithms specified here must match those specified in the manifest file. For every digest algorithm **`XXX`** listed, there must also be a corresponding **`XXX-Digest`** line.

```
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the
 corresponding manifest section)
```

Gives the corresponding digest value for the corresponding manifest section. The value is base-64 encoded. Note that for the purpose of computing the hash of the manifest section, the manifest section starts at the beginning of the opening "**`Name:`**" keyword and continues up to, but not including, the next section's "**`Name:`**" keyword or the end-of-file. Thus the hash includes the blank line(s) at the end of a section and any newline(s) preceding the next "**`Name:`**" keyword or end-of-file.

```
//************************************************************
// Signature Block File Example
//************************************************************
```

A signature block file is a raw binary file (not base-64 encoded) that is a PKCS#7 defined format signature block. The signature block covers exactly the contents of the signer's information file. There must be a correspondence between the name of the signer's information file and the signature block file. The base name matches, and the three-character extension is modified to reflect the signature algorithm used according to the following rules:

- DSA signature algorithm (which uses SHA-1 hash): extension is DSA.
- RSA signature algorithm with MD5 hash: extension is RSA.

So for example with a signer's information file name of "myinfo.SF," the corresponding DSA signature block file name would be "myinfo.DSA."

The format of a signature block file is defined in [PKCS].

```
//*********************************************************
// "X-Intel-BIS-ParameterSet" Attribute value
//   Binary Value of "X-Intel-BIS-ParameterSet" Attribute.
//   (Value is Base-64 encoded in actual signed manifest).
//*********************************************************

#define BOOT_OBJECT_AUTHORIZATION_PARMSET_GUID  \
  {0xedd35e31,0x7b9,0x11d2,0x83,0xa3,0x0,0xa0,0xc9,0x1f,0xad,0xcf}
```

This preprocessor symbol gives the value for an attribute inserted in signed manifests to distinguish updates of BIS parameters from updates of other parameters. The representation inserted into the manifest is base-64 encoded.

Description

This function updates one of the configurable parameters of the Boot Object Authorization set (Boot Object Authorization Certificate or Boot Authorization Check Flag). It passes back a new unique update token that must be included in the request credential for the next update of any parameter in the Boot Object Authorization set. The token value is unique to this platform, parameter set, and instance of parameter values. In particular, the token changes to a new unique value whenever any parameter in this set is changed.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NO_MAPPING	The *AppHandle* parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.
EFI_DEVICE_ERROR	The function encountered an unexpected internal error in a cryptographic software module.
EFI_SECURITY_VIOLATION	The signed manifest supplied as the *RequestCredential* parameter was invalid (could not be parsed), or The signed manifest supplied as the *RequestCredential* parameter failed to verify using the installed Boot Object Authorization Certificate or the signer's Certificate in *RequestCredential*, or Platform-specific authorization failed, or

continued

Status Codes Returned (continued)

EFI_SECURITY_VIOLATION (continued)	The signed manifest supplied as the *RequestCredential* parameter did not include the **X-Intel-BIS-ParameterSet** attribute value, or The **X-Intel-BIS-ParameterSet** attribute value supplied did not match the required GUID value, or The signed manifest supplied as the *RequestCredential* parameter did not include the **X-Intel-BIS-ParameterSetToken** attribute value, or The **X-Intel-BIS-ParameterSetToken** attribute value supplied did not match the platform's current update-token value, or The signed manifest supplied as the *RequestCredential* parameter did not include the **X-Intel-BIS-ParameterId** attribute value, or The **X-Intel-BIS-ParameterId** attribute value supplied did not match one of the permitted values, or The signed manifest supplied as the *RequestCredential* parameter did not include the **X-Intel-BIS-ParameterValue** attribute value, or Any other required attribute value was missing, or The new certificate supplied was too big to store, or The new certificate supplied was invalid (could not be parsed), or The new certificate supplied had an unsupported combination of key algorithm and key length, or The new check flag value supplied is the wrong length (1 byte), or The signed manifest supplied as the *RequestCredential* parameter did not include a signer certificate, or The signed manifest supplied as the *RequestCredential* parameter did not include the manifest section named "**memory:UpdateRequestParameters**," or

continued

Status Codes Returned (continued)

EFI_SECURITY_VIOLATION (continued)	The signed manifest supplied as the *RequestCredential* parameter had a signing certificate with an unsupported public-key algorithm, or The manifest section named "**memory:UpdateRequestParameters**" did not include a digest with a digest algorithm corresponding to the signing certificate's public key algorithm, or The zero-length data object referenced by the manifest section named "**memory:UpdateRequestParameters**" did not verify with the digest supplied in that manifest section, or The signed manifest supplied as the *RequestCredential* parameter did not include a signer's information file with the **SignerInformationName** identifying attribute value "**BIS_UpdateManifestSignerInfoName**," or There were no signers associated with the identified signer's information file, or There was more than one signer associated with the identified signer's information file, or Any other unspecified security violation occurred.
EFI_DEVICE_ERROR	An unexpected internal error occurred while analyzing the new certificate's key algorithm, or An unexpected internal error occurred while attempting to retrieve the public key algorithm of the manifest's signer's certificate, or An unexpected internal error occurred in a cryptographic software module.
EFI_INVALID_PARAMETER	The *RequestCredential* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *RequestCredential.Data* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *NewUpdateToken* parameter supplied by the caller is **NULL** or an invalid memory reference.

EFI_BIS.VerifyBootObject()

Summary

Verifies the integrity and authorization of the indicated data object according to the indicated credentials.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_VERIFY_BOOT_OBJECT)(
  IN  BIS_APPLICATION_HANDLE AppHandle,
  IN  EFI_BIS_DATA           *Credentials,
  IN  EFI_BIS_DATA           *DataObject,
  OUT BOOLEAN                *IsVerified
  );
```

Parameters

AppHandle	An opaque handle that identifies the caller's instance of initialization of the BIS service. Type **BIS_APPLICATION_HANDLE** is defined in the **Initialize()** function description.
Credentials	A Signed Manifest containing verification information for the indicated data object. The Manifest signature itself must meet the requirements described below. This parameter is optional if a Boot Authorization Check is currently not required on this platform (*Credentials.Data* may be **NULL**), otherwise this parameter is required. The required syntax of the Signed Manifest is described in the Related Definition for Manifest Syntax below. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.
DataObject	An in-memory copy of the raw data object to be verified. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.
IsVerified	The function writes **TRUE** if the verification succeeded, otherwise **FALSE**.

Related Definitions

```
//*******************************************************************
// Manifest Syntax
//*******************************************************************
```

The Signed Manifest consists of three parts grouped together into an Electronic Shrink Wrap archive as described in [SM spec]: a manifest file, a signer's information file, and a signature block file. These three parts along with examples are described in the following sections. In these examples, text in parentheses is a description of the text that would appear in the signed manifest. Text outside of parentheses must appear exactly as shown. Also note that manifest files and signer's information files must conform to a 72-byte line-length limit. Continuation lines (lines beginning with a single "space" character) are used for lines longer than 72 bytes. The examples given here follow this rule for continuation lines.

Note that the manifest file and signer's information file parts of a Signed Manifest are ASCII (not Unicode) text files. In cases where these files contain a base-64 encoded string, the string is an ASCII (not Unicode) string before base-64 encoding.

```
//*******************************************************************
// Manifest File Example
//*******************************************************************
```

The manifest file must include a section referring to a memory-type data object with the reserved name as shown in the example below. This data object is the Boot Object to be verified. An example manifest file is shown below.

```
Manifest-Version: 2.0
ManifestPersistentId: (base-64 representation of a unique GUID)

Name: memory:BootObject
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the
 boot object)
```

A line-by-line description of this manifest file is as follows.

```
Manifest-Version: 2.0
```

This is a standard header line that all signed manifests have. It must appear exactly as shown.

```
ManifestPersistentId: (base-64 representation of a unique GUID)
```

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every manifest file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

```
 Name: memory:BootObject
```

This identifies the section that carries the integrity data for the Boot Object. The string "**`memory:BootObject`**" must appear exactly as shown. Note that the Boot Object cannot be found directly from this manifest. A caller verifying the Boot Object integrity must load the Boot Object into memory and specify its memory location explicitly to this verification function through the *`DataObject`* parameter.

```
Digest-Algorithms: SHA-1
```

This enumerates the digest algorithms for which integrity data is included for the data object. For systems with DSA signing, SHA-1 hash, and 1024-bit key length, the digest algorithm must be "**`SHA-1`**." For systems with RSA signing, MD5 hash, and 512-bit key length, the digest algorithm must be "**`MD5`**." Multiple algorithms can be specified as a whitespace-separated list. For every digest algorithm **`XXX`** listed, there must also be a corresponding **`XXX-Digest`** line.

```
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the boot object)
```

Gives the corresponding digest value for the data object. The value is base-64 encoded.

```
//*************************************************************
// Signer's Information File Example
//*************************************************************
```

The signer's information file must include a section whose name matches the reserved data object section name of the section in the Manifest file. This section in the signer's information file carries the integrity data for the corresponding section in the manifest file. An example signer's information file is shown below.

```
Signature-Version: 2.0
SignerInformationPersistentId: (base-64 representation of a
 unique GUID)
SignerInformationName: BIS_VerifiableObjectSignerInfoName

Name: memory:BootObject
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the
 corresponding manifest section)
```

A line-by-line description of this signer's information file is as follows.

```
Signature-Version: 2.0
```

This is a standard header line that all signed manifests have. It must appear exactly as shown.

```
SignerInformationPersistentId: (base-64 representation of a unique GUID)
```

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every signer's information file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

```
SignerInformationName: BIS_VerifiableObjectSignerInfoName
```

The left-hand string must appear exactly as shown. The right-hand string must appear exactly as shown.

```
Name: memory:BootObject
```

This identifies the section in the signer's information file corresponding to the section with the same name in the manifest file described earlier. The string "**`memory:BootObject`**" must appear exactly as shown.

```
Digest-Algorithms: SHA-1
```

This enumerates the digest algorithms for which integrity data is included for the corresponding manifest section. Strings identifying digest algorithms are the same as in the manifest file. The digest algorithms specified here must match those specified in the manifest file. For every digest algorithm **`XXX`** listed, there must also be a corresponding **`XXX-Digest`** line.

```
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the
corresponding manifest section)
```

Gives the corresponding digest value for the corresponding manifest section. The value is base-64 encoded. Note that for the purpose of computing the hash of the manifest section, the manifest section starts at the beginning of the opening "**`Name:`**" keyword and continues up to, but not including, the next section's "**`Name:`**" keyword or the end-of-file. Thus the hash includes the blank line(s) at the end of a section and any newline(s) preceding the next "**`Name:`**" keyword or end-of-file.

```
//***********************************************************
// Signature Block File Example
//***********************************************************
```

A signature block file is a raw binary file (not base-64 encoded) that is a PKCS#7 defined format signature block. The signature block covers exactly the contents of the signer's information file. There must be a correspondence between the name of the signer's information file and the signature block file. The base name matches, and the three-character extension is modified to reflect the signature algorithm used according to the following rules:

- DSA signature algorithm (which uses SHA-1 hash): extension is DSA.
- RSA signature algorithm with MD5 hash: extension is RSA.

So for example with a signer's information file name of "myinfo.SF," the corresponding DSA signature block file name would be "myinfo.DSA."

The format of a signature block file is defined in [PKCS].

Description

This function verifies the integrity and authorization of the indicated data object according to the indicated credentials. The rules for successful verification depend on whether or not a Boot Authorization Check is currently required on this platform.

If a Boot Authorization Check is *not* currently required on this platform, no authorization check is performed. However, the following rules are applied for an integrity check:

- In this case, the credentials are optional. If they are *not* supplied (`Credentials.Data` is **`NULL`**), no integrity check is performed, and the function returns immediately with a "success" indication and `IsVerified` is **`TRUE`**.
- If the credentials *are* supplied (`Credentials.Data` is other than **`NULL`**), integrity checks are performed as follows:
  - Verify the credentials – The credentials parameter is a valid signed Manifest, with a single signer. The signer's identity is included in the credential as a certificate.
  - Verify the data object – The Manifest must contain a section named "**`memory:BootObject`**," with associated verification information (in other words, hash value). The hash value from this Manifest section must match the hash value computed over the specified `DataObject` data.
  - If these checks succeed, the function returns with a "success" indication and `IsVerified` is **`TRUE`**. Otherwise, `IsVerified` is **`FALSE`** and the function returns with a "security violation" indication.

If a Boot Authorization Check *is* currently required on this platform, authorization and integrity checks are performed. The integrity check is the same as in the case above, except that it is required. The following rules are applied:

- Verify the credentials – The credentials parameter is required in this case (`Credentials.Data` must be other than **`NULL`**). The credentials parameter is a valid Signed Manifest, with a single signer. The signer's identity is included in the credential as a certificate.
- Verify the data object – The Manifest must contain a section named "**`memory:BootObject`**," with associated verification information (in other words, hash value). The hash value from this Manifest section must match the hash value computed over the specified `DataObject` data.
- Do Authorization check – This happens one of two ways depending on whether or not the platform currently has a Boot Object Authorization Certificate configured.
  - If a Boot Object Authorization Certificate is not currently configured, this function interacts with the user in a platform-specific way to determine whether the operation should succeed.
  - If a Boot Object Authorization Certificate *is* currently configured, this function uses the Boot Object Authorization Certificate to determine whether the operation should succeed. The public key certified by the signer's certificate must match the public key in the Boot Object Authorization Certificate configured for this platform. The match must be direct, that is, the signature authority cannot be delegated along a certificate chain.

— If these checks succeed, the function returns with a "success" indication and *IsVerified* is **TRUE**. Otherwise, *IsVerified* is **FALSE** and the function returns with a "security violation" indication.

Note that if a Boot Authorization Check is currently required on this platform this function *always* performs an authorization check, either through platform-specific user interaction or through a signature generated with the private key corresponding to the public key in the platform's Boot Object Authorization Certificate.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NO_MAPPING	The *AppHandle* parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.
EFI_INVALID_PARAMETER	The *Credentials* parameter supplied by the caller is **NULL** or an invalid memory reference, or The Boot Authorization Check is currently required on this platform and the *Credentials.Data* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *DataObject* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *DataObject.Data* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *IsVerified* parameter supplied by the caller is **NULL** or an invalid memory reference.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.
EFI_SECURITY_VIOLATION	The signed manifest supplied as the *Credentials* parameter was invalid (could not be parsed), or The signed manifest supplied as the *Credentials* parameter failed to verify using the installed Boot Object Authorization Certificate or the signer's Certificate in *Credentials*, or Platform-specific authorization failed, or Any other required attribute value was missing, or The signed manifest supplied as the *Credentials* parameter did not include a signer certificate, or

continued

Status Codes Returned (continued)

EFI_SECURITY_VIOLATION (continued)	The signed manifest supplied as the *Credentials* parameter did not include the manifest section named "**memory:BootObject**," or The signed manifest supplied as the *Credentials* parameter had a signing certificate with an unsupported public-key algorithm, or The manifest section named "**memory:BootObject**" did not include a digest with a digest algorithm corresponding to the signing certificate's public key algorithm, or The data object supplied as the *DataObject* parameter and referenced by the manifest section named "**memory:BootObject**" did not verify with the digest supplied in that manifest section, or The signed manifest supplied as the *Credentials* parameter did not include a signer's information file with the **SignerInformationName** identifying attribute value "**BIS_VerifiableObjectSignerInfoName**," or There were no signers associated with the identified signer's information file, or There was more than one signer associated with the identified signer's information file, or The platform's check flag is "on" (requiring authorization checks) but the *Credentials.Data* supplied by the caller is **NULL**, or Any other unspecified security violation occurred.
EFI_DEVICE_ERROR	An unexpected internal error occurred while attempting to retrieve the public key algorithm of the manifest's signer's certificate, or An unexpected internal error occurred in a cryptographic software module.

EFI_BIS.VerifyObjectWithCredential()

Summary

Verifies the integrity and authorization of the indicated data object according to the indicated credentials and authority certificate.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_VERIFY_OBJECT_WITH_CREDENTIAL)(
  IN  BIS_APPLICATION_HANDLE AppHandle,
  IN  EFI_BIS_DATA           *Credentials,
  IN  EFI_BIS_DATA           *DataObject,
  IN  EFI_BIS_DATA           *SectionName,
  IN  EFI_BIS_DATA           *AuthorityCertificate,
  OUT BOOLEAN                *IsVerified
  );
```

Parameters

AppHandle	An opaque handle that identifies the caller's instance of initialization of the BIS service. Type **BIS_APPLICATION_HANDLE** is defined in the **Initialize()** function description.
Credentials	A Signed Manifest containing verification information for the indicated data object. The Manifest signature itself must meet the requirements described below. The required syntax of the Signed Manifest is described in the Related Definition of Manifest Syntax below. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.
DataObject	An in-memory copy of the raw data object to be verified. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.
SectionName	An ASCII (not Unicode) string giving the section name in the manifest holding the verification information (in other words, hash value) that corresponds to *DataObject*. Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.

AuthorityCertificate	A digital certificate whose public key must match the signer's public key which is found in the credentials. This parameter is optional (*AuthorityCertificate.Data* may be **NULL**). Type **EFI_BIS_DATA** is defined in the **Initialize()** function description.
IsVerified	The function writes **TRUE** if the verification was successful. Otherwise, the function writes **FALSE**.

Related Definitions

```
//************************************************************
// Manifest Syntax
//************************************************************
```

The Signed Manifest consists of three parts grouped together into an Electronic Shrink Wrap archive as described in [SM spec]: a manifest file, a signer's information file, and a signature block file. These three parts along with examples are described in the following sections. In these examples, text in parentheses is a description of the text that would appear in the signed manifest. Text outside of parentheses must appear exactly as shown. Also note that manifest files and signer's information files must conform to a 72-byte line-length limit. Continuation lines (lines beginning with a single "space" character) are used for lines longer than 72 bytes. The examples given here follow this rule for continuation lines.

Note that the manifest file and signer's information file parts of a Signed Manifest are ASCII (not Unicode) text files. In cases where these files contain a base-64 encoded string, the string is an ASCII (not Unicode) string before base-64 encoding.

```
//************************************************************
// Manifest File Example
//************************************************************
```

The manifest file must include a section referring to a memory-type data object with the caller-chosen name as shown in the example below. This data object is the Data Object to be verified. An example manifest file is shown below.

```
Manifest-Version: 2.0
ManifestPersistentId: (base-64 representation of a unique GUID)

Name: (a memory-type data object name)
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the
 data object)
```

A line-by-line description of this manifest file is as follows.

```
Manifest-Version: 2.0
```

This is a standard header line that all signed manifests have. It must appear exactly as shown.

```
ManifestPersistentId: (base-64 representation of a unique GUID)
```

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every manifest file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

```
Name: (a memory-type data object name)
```

This identifies the section that carries the integrity data for the target Data Object. The right-hand string must obey the syntax for memory-type references, that is, it is of the form "**`memory:SomeUniqueName`**." The "**`memory:`**" part of this string must appear exactly. The "**`SomeUniqueName`**" part is chosen by the caller. It must be unique within the section names in this manifest file. The entire "**`memory:SomeUniqueName`**" string must match exactly the corresponding string in the signer's information file described below. Furthermore, this entire string must match the value given for the *`SectionName`* parameter to this function. Note that the target Data Object cannot be found directly from this manifest. A caller verifying the Data Object integrity must load the Data Object into memory and specify its memory location explicitly to this verification function through the *`DataObject`* parameter.

```
Digest-Algorithms: SHA-1
```

This enumerates the digest algorithms for which integrity data is included for the data object. For systems with DSA signing, SHA-1 hash, and 1024-bit key length, the digest algorithm must be "**`SHA-1`**." For systems with RSA signing, MD5 hash, and 512-bit key length, the digest algorithm must be "**`MD5`**." Multiple algorithms can be specified as a whitespace-separated list. For every digest algorithm **`XXX`** listed, there must also be a corresponding **`XXX-Digest`** line.

```
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the data object)
```

Gives the corresponding digest value for the data object. The value is base-64 encoded.

```
//************************************************************
// Signer's Information File Example
//************************************************************
```

The signer's information file must include a section whose name matches the reserved data object section name of the section in the Manifest file. This section in the signer's information file carries the integrity data for the corresponding section in the manifest file. An example signer's information file is shown below.

```
Signature-Version: 2.0
SignerInformationPersistentId: (base-64 representation of a
unique GUID)
SignerInformationName: BIS_VerifiableObjectSignerInfoName

Name: (a memory-type data object name)
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the
corresponding manifest section)
```

A line-by-line description of this signer's information file is as follows.

```
Signature-Version: 2.0
```

This is a standard header line that all signed manifests have. It must appear exactly as shown.

```
SignerInformationPersistentId: (base-64 representation of a unique GUID)
```

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every signer's information file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

```
SignerInformationName: BIS_VerifiableObjectSignerInfoName
```

The left-hand string must appear exactly as shown. The right-hand string must appear exactly as shown.

```
Name: (a memory-type data object name)
```

This identifies the section in the signer's information file corresponding to the section with the same name in the manifest file described earlier. The right-hand string must match exactly the corresponding string in the manifest file described above.

```
Digest-Algorithms: SHA-1
```

This enumerates the digest algorithms for which integrity data is included for the corresponding manifest section. Strings identifying digest algorithms are the same as in the manifest file. The digest algorithms specified here must match those specified in the manifest file. For every digest algorithm **`XXX`** listed, there must also be a corresponding **`XXX-Digest`** line.

```
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the
corresponding manifest section)
```

Gives the corresponding digest value for the corresponding manifest section. The value is base-64 encoded. Note that for the purpose of computing the hash of the manifest section, the manifest section starts at the beginning of the opening "**`Name:`**" keyword and continues up to, but not including, the next section's "**`Name:`**" keyword or the end-of-file. Thus the hash includes the blank line(s) at the end of a section and any newline(s) preceding the next "**`Name:`**" keyword or end-of-file.

```
//*************************************************************
// Signature Block File Example
//*************************************************************
```

A signature block file is a raw binary file (not base-64 encoded) that is a PKCS#7 defined format signature block. The signature block covers exactly the contents of the signer's information file. There must be a correspondence between the name of the signer's information file and the signature block file. The base name matches, and the three-character extension is modified to reflect the signature algorithm used according to the following rules:

- DSA signature algorithm (which uses SHA-1 hash): extension is DSA.
- RSA signature algorithm with MD5 hash: extension is RSA.

So for example with a signer's information file name of "myinfo.SF," the corresponding DSA signature block file name would be "myinfo.DSA."

The format of a signature block file is defined in [PKCS].

Description

This function verifies the integrity and authorization of the indicated data object according to the indicated credentials and authority certificate.

Both an integrity check and an authorization check are performed. The rules for a successful integrity check are:

- Verify the credentials – The credentials parameter is a valid Signed Manifest, with a single signer. The signer's identity is included in the credential as a certificate.
- Verify the data object – The Manifest must contain a section with the name as specified by the *SectionName* parameter, with associated verification information (in other words, hash value). The hash value from this Manifest section must match the hash value computed over the data specified by the *DataObject* parameter of this function.

The authorization check is optional. It is performed only if the *AuthorityCertificate.Data* parameter is other than **NULL**. If it is other than **NULL**, the rules for a successful authorization check are:

- The *AuthorityCertificate* parameter is a valid digital certificate. There is no requirement regarding the signer (issuer) of this certificate.
- The public key certified by the signer's certificate must match the public key in the *AuthorityCertificate*. The match must be direct, that is, the signature authority cannot be delegated along a certificate chain.

If all of the integrity and authorization check rules are met, the function returns with a "success" indication and *IsVerified* is **TRUE**. Otherwise, it returns with a nonzero specific error code and *IsVerified* is **FALSE**.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NO_MAPPING	The *AppHandle* parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.
EFI_INVALID_PARAMETER	The *Credentials* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *Credentials.Data* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *Credentials.Length* supplied by the caller is zero, or The *DataObject* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *DataObject.Data* parameter supplied by the caller is **NULL** or an invalid memory reference, or

continued

Status Codes Returned (continued)

EFI_INVALID_PARAMETER (continued)	The *SectionName* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *SectionName.Data* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *SectionName.Length* supplied by the caller is zero, or The *AuthorityCertificate* parameter supplied by the caller is **NULL** or an invalid memory reference, or The *IsVerified* parameter supplied by the caller is **NULL** or an invalid memory reference.
EFI_OUT_OF_RESOURCES	The function failed due to lack of memory or other resources.
EFI_SECURITY_VIOLATION	The *Credentials.Data* supplied by the caller is **NULL**, or The *AuthorityCertificate* supplied by the caller was invalid (could not be parsed), or The signed manifest supplied as *Credentials* failed to verify using the *AuthorityCertificate* supplied by the caller or the manifest's signer's certificate, or Any other required attribute value was missing, or The signed manifest supplied as the *Credentials* parameter did not include a signer certificate, or The signed manifest supplied as the *Credentials* parameter did not include the manifest section named according to *SectionName*, or The signed manifest supplied as the *Credentials* parameter had a signing certificate with an unsupported public-key algorithm, or The manifest section named according to *SectionName* did not include a digest with a digest algorithm corresponding to the signing certificate's public key algorithm, or The data object supplied as the *DataObject* parameter and referenced by the manifest section named according to *SectionName* did not verify with the digest supplied in that manifest section, or

continued

Status Codes Returned (continued)

EFI_SECURITY_VIOLATION (continued)	The signed manifest supplied as the *`Credentials`* parameter did not include a signer's information file with the **`SignerInformationName`** identifying attribute value "**`BIS_VerifiableObjectSignerInfoName`**," or There were no signers associated with the identified signer's information file, or There was more than one signer associated with the identified signer's information file, or Any other unspecified security violation occurred.
EFI_DEVICE_ERROR	An unexpected internal error occurred while attempting to retrieve the public key algorithm of the manifest's signer's certificate, or An unexpected internal error occurred in a cryptographic software module.

16
Protocols - Debugger Support

This chapter describes a minimal set of protocols and associated data structures necessary to enable the creation of source level debuggers for EFI. It does not fully define a debugger design. Using the services described in this document, it should also be possible to implement a variety of debugger solutions.

16.1 Overview

Efficient EFI driver and application development requires the availability of source level debugging facilities. Although completely on-target debuggers are clearly possible, EFI debuggers are generally expected to be remotely hosted. That is to say, the debugger itself will be split between two machines, which are the host and target. A majority of debugger code runs on the host that is typically responsible for disassembly, symbol management, source display, and user interface. Similarly, a smaller piece of code runs on the target that establishes the communication to the host and proxies requests from the host. The on-target code is known as the “debug agent.”

The debug agent design is subdivided further into two parts, which are the processor/platform abstraction and the debugger host specific communication grammar. This specification describes architectural interfaces for the former only. Specific implementations for various debugger host communication grammars can be created that make use of the facilities described in this specification.

The processor/platform abstraction is presented as a pair of protocol interfaces, which are the Debug Support protocol and the Debug Port protocol.

The Debug Support protocol abstracts the processor’s debugging facilities, namely a mechanism to manage the processor’s context via caller-installable exception handlers.

The Debug Port protocol abstracts the device that is used for communication between the host and target. Typically this will be a 16550 serial port, 1394 device, or other device that is nominally a serial stream.

Furthermore, a table driven, quiescent, memory-only mechanism for determining the base address of PE32+ images is provided to enable the debugger host to determine where images are located in memory.

Aside from timing differences that occur because of running code associated with the debug agent and user initiated changes to the machine context, the operation of the on-target debugger component must be transparent to the rest of the system. In addition, no portion of the debug agent that runs in interrupt context may make any calls to EFI services or other protocol interfaces.

The services described in this document do not comprise a complete debugger, rather they provide a minimal abstraction required to implement a wide variety of debugger solutions.

16.2 EFI Debug Support Protocol

This section defines the EFI Debug Support protocol which is used by the debug agent.

16.2.1 EFI Debug Support Protocol Overview

The debug-agent needs to be able to gain control of the machine when certain types of events occur; i.e. breakpoints, processor exceptions, etc. Additionally, the debug agent must also be able to periodically gain control during operation of the machine to check for asynchronous commands from the host. The EFI Debug Support protocol services enable these capabilities.

The EFI Debug Support protocol interfaces produce callback registration mechanisms which are used by the debug agent to register functions that are invoked either periodically or when specific processor exceptions. When they are invoked by the Debug Support driver, these callback functions are passed the current machine context record. The debug agent may modify this context record to change the machine context which is restored to the machine after the callback function returns. The debug agent does not run in the same context as the rest of EFI and all modifications to the machine context are deferred until after the callback function returns.

It is expected that there will typically be two instances of the EFI Debug Support protocol in the system. On associated with the native processor instruction set (IA-32 or Itanium processor family), and one for the EFI virtual machine that implements EFI byte code (EBC).

While multiple instances of the EFI Debug Support protocol are expected, there must never be more than one for any given instruction set.

EFI_DEBUG_SUPPORT_PROTOCOL

Summary

This protocol provides the services to allow the debug agent to register callback functions that are called either periodically or when specific processor exceptions occur.

GUID

```
#define EFI_DEBUG_SUPPORT_PROTOCOL_GUID  \
{0x2755590C,0x6F3C,0x42FA,0x9E,0xA4,0xA3,0xBA,0x54,0x3C,0xDA,0x25}
```

Protocol Interface Structure

```
typedef struct {
  EFI_INSTRUCTION_SET_ARCHITECTURE     Isa;
  EFI_GET_MAXIMUM_PROCESSOR_INDEX      GetMaximumProcessorIndex;
  EFI_REGISTER_PERIODIC_CALLBACK       RegisterPeriodicCallback;
  EFI_REGISTER_EXCEPTION_CALLBACK      RegisterExceptionCallback;
  EFI_INVALIDATE_INSTRUCTION_CACHE     InvalidateInstructionCache;
} EFI_DEBUG_SUPPORT_PROTOCOL;
```

Parameters

Isa Declares the processor architecture for this instance of the EFI Debug Support protocol.

GetMaximumProcessorIndex

Returns the maximum processor index value that may be used with **RegisterPeriodicCallback()** and **RegisterExceptionCallback()**. See the **GetMaximumProcessorIndex()** function description.

RegisterPeriodicCallback

Registers a callback function that will be invoked periodically and asynchronously to the execution of EFI. See the **RegisterPeriodicCallback()** function description.

RegisterExceptionCallback

Registers a callback function that will be called each time the specified processor exception occurs. See the **RegisterExceptionCallback()** function description.

InvalidateInstructionCache	Invalidate the instruction cache of the processor. This is required by processor architectures where instruction and data caches are not coherent when instructions in the code under debug has been modified by the debug agent. See the **InvalidateInstructionCache()** function description.

Related Definitions

Refer to the Microsoft PE/COFF Specification revision 6.2 or later for IMAGE_FILE_MACHINE definitions.

NOTE

At the time of publication of this specification, the latest revision of the PE/COFF specification was 6.2. The definition of IMAGE_FILE_MACHINE_EBC is not included in revision 6.2 of the PE/COFF specification. It will be added in a future revision of the PE/COFF specification.

```
typedef enum {
  IsaIa32 = IMAGE_FILE_MACHINE_I386,   // 0x014C
  IsaIpf  = IMAGE_FILE_MACHINE_IA64,   // 0x0200
  IsaEbc  = IMAGE_FILE_MACHINE_EBC     // 0x0EBC
} EFI_INSTRUCTION_SET_ARCHITECTURE
```

Description

The EFI Debug Support protocol provides the interfaces required to register debug agent callback functions and to manage the processor's instruction stream as required. Registered callback functions are invoked in interrupt context when the specified event occurs.

The driver that produces the EFI Debug Support protocol is also responsible for saving the machine context prior to invoking a registered callback function and restoring it after the callback function returns prior to returning to the code under debug. If the debug agent has modified the context record, the modified context must be used in the restore operation.

Furthermore, if the debug agent modifies any of the code under debug (to set a software breakpoint for example), it must call the **InvalidateInstructionCache()** function for the region of memory that has been modified.

EFI_DEBUG_SUPPORT_PROTOCOL.GetMaximumProcessorIndex()

Summary

Returns the maximum value that may be used for the *ProcessorIndex* parameter in **RegisterPeriodicCallback()** and **RegisterExceptionCallback()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_MAXIMUM_PROCESSOR_INDEX) (
  IN EFI_DEBUG_SUPPORT_PROTOCOL   *This,
  OUT UINTN                       *MaxProcessorIndex
  );
```

Parameters

This	A pointer to the **EFI_DEBUG_SUPPORT_PROTOCOL** instance. Type **EFI_DEBUG_SUPPORT_PROTOCOL** is defined in Section 16.2.
MaxProcessorIndex	Pointer to a caller-allocated **UINTN** in which the maximum supported processor index is returned.

Description

The **GetMaximumProcessorIndex()** function returns the maximum processor index in the output parameter *MaxProcessorIndex*. This value is the largest value that may be used in the *ProcessorIndex* parameter for both **RegisterPeriodicCallback()** and **RegisterExceptionCallback()**. All values between 0 and *MaxProcessorIndex* must be supported by **RegisterPeriodicCallback()** and **RegisterExceptionCallback()**.

It is the responsibility of the caller to insure all parameters are correct. There is no provision for parameter checking by **GetMaximumProcessorIndex()**. The implementation behavior when an invalid parameter is passed is not defined by this specification.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.

EFI_DEBUG_SUPPORT_PROTOCOL.RegisterPeriodicCallback()

Summary

Registers a function to be called back periodically in interrupt context.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_REGISTER_PERIODIC_CALLBACK) (
  IN EFI_DEBUG_SUPPORT_PROTOCOL   *This,
  IN UINTN                        ProcessorIndex,
  IN EFI_PERIODIC_CALLBACK        PeriodicCallback
  );
```

Parameters

This	A pointer to the **EFI_DEBUG_SUPPORT_PROTOCOL** instance. Type **EFI_DEBUG_SUPPORT_PROTOCOL** is defined in Section 16.2.
ProcessorIndex	Specifies which processor the callback function applies to.
PeriodicCallback	A pointer to a function of type **PERIODIC_CALLBACK** that is the main periodic entry point of the debug agent. It receives as a parameter a pointer to the full context of the interrupted execution thread.

Related Definitions

```
typedef
VOID (*EFI_PERIODIC_CALLBACK) (
  IN OUT EFI_SYSTEM_CONTEXT      SystemContext
  );

typedef union {
  EFI_SYSTEM_CONTEXT_EBC        *SystemContextEbc,
  EFI_SYSTEM_CONTEXT_IA32       *SystemContextIa32,
  EFI_SYSTEM_CONTEXT_IPF        *SystemContextIpf
} EFI_SYSTEM_CONTEXT;

// System context for virtual EBC processors
typedef struct {
  UINT64             R0, R1, R2, R3, R4, R5, R6, R7;
  UINT64             Flags;
  UINT64             ControlFlags;
  UINT64             Ip;
} EFI_SYSTEM_CONTEXT_EBC;
```

NOTE

When the context record field is larger than the register being stored in it, the upper bits of the context record field are unused and ignored.

```
// System context for IA-32 processors
typedef struct {
  UINT32              ExceptionData;   // ExceptionData is
                                       // additional data pushed
                                       // on the stack by some
                                       // types of IA-32
                                       // exceptions
  EFI_FXSAVE_STATE    FxSaveState;
  UINT32              Dr0, Dr1, Dr2, Dr3, Dr6, Dr7;
  UINT32              Cr0, Cr1 /* Reserved */, Cr2, Cr3, Cr4;
  UINT32              Eflags;
  UINT32              Ldtr, Tr;
  UINT32              Gdtr[2], Idtr[2];
  UINT32              Eip;
  UINT32              Gs, Fs, Es, Ds, Cs, Ss;
  UINT32              Edi, Esi, Ebp, Esp, Ebx, Edx, Ecx, Eax;
} EFI_SYSTEM_CONTEXT_IA32;

// FXSAVE_STATE - FP / MMX / XMM registers
typedef struct {
  UINT16              Fcw;
  UINT16              Fsw;
  UINT16              Ftw;
  UINT16              Opcode;
  UINT32              Eip;
  UINT16              Cs;
  UINT16              Reserved1;
  UINT32              DataOffset;
  UINT16              Ds;
  UINT8               Reserved2[10];
  UINT8               St0Mm0[10], Reserved3[6];
  UINT8               St0Mm1[10], Reserved4[6];
  UINT8               St0Mm2[10], Reserved5[6];
  UINT8               St0Mm3[10], Reserved6[6];
  UINT8               St0Mm4[10], Reserved7[6];
  UINT8               St0Mm5[10], Reserved8[6];
  UINT8               St0Mm6[10], Reserved9[6];
  UINT8               St0Mm7[10], Reserved10[6];
  UINT8               Reserved11[22 * 16];
} EFI_FXSAVE_STATE;
```

```
// System context for Itanium processor family
typedef struct {
  UINT64    Reserved;

  UINT64     R1,  R2,  R3,  R4,  R5,  R6,  R7,  R8,  R9, R10,
            R11, R12, R13, R14, R15, R16, R17, R18, R19, R20,
            R21, R22, R23, R24, R25, R26, R27, R28, R29, R30,
            R31;

  UINT64     F2[2],  F3[2],  F4[2],  F5[2],  F6[2],
             F7[2],  F8[2],  F9[2], F10[2], F11[2],
            F12[2], F13[2], F14[2], F15[2], F16[2],
            F17[2], F18[2], F19[2], F20[2], F21[2],
            F22[2], F23[2], F24[2], F25[2], F26[2],
            F27[2], F28[2], F29[2], F30[2], F31[2];

  UINT64    Pr;

  UINT64    B0, B1, B2, B3, B4, B5, B6, B7;

  // application registers
  UINT64    ArRsc, ArBsp, ArBspstore, ArRnat;
  UINT64    ArFcr;
  UINT64    ArEflag, ArCsd, ArSsd, ArCflg;
  UINT64    ArFsr, ArFir, ArFdr;
  UINT64    ArCcv;
  UINT64    ArUnat;
  UINT64    ArFpsr;
  UINT64    ArPfs, ArLc, ArEc;

  // control registers
  UINT64    CrDcr, CrItm, CrIva, CrPta, CrIpsr, CrIsr;
  UINT64    CrIip, CrIfa, CrItir, CrIipa, CrIfs, CrIim;
  UINT64    CrIha;

  // debug registers
  UINT64    Dbr0, Dbr1, Dbr2, Dbr3, Dbr4, Dbr5, Dbr6, Dbr7;
  UINT64    Ibr0, Ibr1, Ibr2, Ibr3, Ibr4, Ibr5, Ibr6, Ibr7;

  // virtual registers
  UINT64    IntNat;    // nat bits for R1-R31

} EFI_SYSTEM_CONTEXT_IPF;
```

Description

The **`RegisterPeriodicCallback()`** function registers and enables the on-target debug agent's periodic entry point. To unregister and disable calling the debug agent's periodic entry point, call **`RegisterPeriodicCallback()`** passing a **`NULL`** *`PeriodicCallback`* parameter.

The implementation must handle saving and restoring the processor context to/from the system context record around calls to the registered callback function.

If the interrupt is also used by the firmware for the EFI time base or some other use, two rules must be observed. First, the registered callback function must be called before any EFI processing takes place. Second, the Debug Support implementation must perform the necessary steps to pass control to the firmware's corresponding interrupt handler in a transparent manner.

There is no quality of service requirement or specification regarding the frequency of calls to the registered *`PeriodicCallback`* function. This allows the implementation to mitigate a potential adverse impact to EFI timer based services due to the latency induced by the context save/restore and the associated callback function.

It is the responsibility of the caller to insure all parameters are correct. There is no provision for parameter checking by **`RegisterPeriodicCallback()`**. The implementation behavior when an invalid parameter is passed is not defined by this specification.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_ALREADY_STARTED	Non-**`NULL`** *`PeriodicCallback`* parameter when a callback function was previously registered.
EFI_OUT_OF_RESOURCES	System has insufficient memory resources to register new callback function.

EFI_DEBUG_SUPPORT_PROTOCOL.RegisterExceptionCallback()

Summary

Registers a function to be called when a given processor exception occurs.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *REGISTER_EXCEPTION_CALLBACK) (
  IN EFI_DEBUG_SUPPORT_PROTOCOL   *This,
  IN UINTN                        ProcessorIndex,
  IN EFI_EXCEPTION_CALLBACK       ExceptionCallback,
  IN EFI_EXCEPTION_TYPE           ExceptionType
  );
```

Parameters

This	A pointer to the **EFI_DEBUG_SUPPORT_PROTOCOL** instance. Type **EFI_DEBUG_SUPPORT_PROTOCOL** is defined in Section 16.2.
ProcessorIndex	Specifies which processor the callback function applies to.
ExceptionCallback	A pointer to a function of type **EXCEPTION_CALLBACK** that is called when the processor exception specified by *ExceptionType* occurs. Passing **NULL** unregisters any previously registered function associated with *ExceptionType*.
ExceptionType	Specifies which processor exception to hook.

Related Definitions

```
typedef
VOID (*EFI_EXCEPTION_CALLBACK) (
  IN EFI_EXCEPTION_TYPE           ExceptionType,
  IN OUT EFI_SYSTEM_CONTEXT       SystemContext
  );

typedef INTN EFI_EXCEPTION_TYPE;

// EBC Exception types
#define EXCEPT_EBC_UNDEFINED                          0
#define EXCEPT_EBC_DIVIDE_ERROR                       1
#define EXCEPT_EBC_DEBUG                              2
#define EXCEPT_EBC_BREAKPOINT                         3
#define EXCEPT_EBC_OVERFLOW                           4
#define EXCEPT_EBC_INVALID_OPCODE                     5
```

```
#define EXCEPT_EBC_STACK_FAULT                          6
#define EXCEPT_EBC_ALIGNMENT_CHECK                      7
#define EXCEPT_EBC_INSTRUCTION_ENCODING                 8
#define EXCEPT_EBC_BAD_BREAK                            9
#define EXCEPT_EBC_SINGLE_STEP                         10

// IA-32 Exception types
#define EXCEPT_IA32_DIVIDE_ERROR                        0
#define EXCEPT_IA32_DEBUG                               1
#define EXCEPT_IA32_NMI                                 2
#define EXCEPT_IA32_BREAKPOINT                          3
#define EXCEPT_IA32_OVERFLOW                            4
#define EXCEPT_IA32_BOUND                               5
#define EXCEPT_IA32_INVALID_OPCODE                      6
#define EXCEPT_IA32_DOUBLE_FAULT                        8
#define EXCEPT_IA32_INVALID_TSS                        10
#define EXCEPT_IA32_SEG_NOT_PRESENT                    11
#define EXCEPT_IA32_STACK_FAULT                        12
#define EXCEPT_IA32_GP_FAULT                           13
#define EXCEPT_IA32_PAGE_FAULT                         14
#define EXCEPT_IA32_FP_ERROR                           16
#define EXCEPT_IA32_ALIGNMENT_CHECK                    17
#define EXCEPT_IA32_MACHINE_CHECK                      18
#define EXCEPT_IA32_SIMD                               19

// Itanium Processor Family Exception types
#define    EXCEPT_IPF_VHTP_TRANSLATION                  0
#define    EXCEPT_IPF_INSTRUCTION_TLB                   1
#define    EXCEPT_IPF_DATA_TLB                          2
#define    EXCEPT_IPF_ALT_INSTRUCTION_TLB               3
#define    EXCEPT_IPF_ALT_DATA_TLB                      4
#define    EXCEPT_IPF_DATA_NESTED_TLB                   5
#define    EXCEPT_IPF_INSTRUCTION_KEY_MISSED            6
#define    EXCEPT_IPF_DATA_KEY_MISSED                   7
#define    EXCEPT_IPF_DIRTY_BIT                         8
#define    EXCEPT_IPF_INSTRUCTION_ACCESS_BIT            9
#define    EXCEPT_IPF_DATA_ACCESS_BIT                  10
#define    EXCEPT_IPF_BREAKPOINT                       11
#define    EXCEPT_IPF_EXTERNAL_INTERRUPT               12
// 13 - 19 reserved
#define    EXCEPT_IPF_PAGE_NOT_PRESENT                 20
#define    EXCEPT_IPF_KEY_PERMISSION                   21
#define    EXCEPT_IPF_INSTRUCTION_ACCESS_RIGHTS        22
#define    EXCEPT_IPF_DATA_ACCESS_RIGHTS               23
#define    EXCEPT_IPF_GENERAL_EXCEPTION                24
#define    EXCEPT_IPF_DISABLED_FP_REGISTER             25
#define    EXCEPT_IPF_NAT_CONSUMPTION                  26
#define    EXCEPT_IPF_SPECULATION                      27
// 28 reserved
```

```
#define    EXCEPT_IPF_DEBUG                        29
#define    EXCEPT_IPF_UNALIGNED_REFERENCE          30
#define    EXCEPT_IPF_UNSUPPORTED_DATA_REFERENCE   31
#define    EXCEPT_IPF_FP_FAULT                     32
#define    EXCEPT_IPF_FP_TRAP                      33
#define    EXCEPT_IPF_LOWER_PRIVILEGE_TRANSFER_TRAP 34
#define    EXCEPT_IPF_TAKEN_BRANCH                 35
#define    EXCEPT_IPF_SINGLE_STEP                  36
// 37 - 44 reserved
#define    EXCEPT_IPF_IA32_EXCEPTION               45
#define    EXCEPT_IPF_IA32_INTERCEPT               46
#define    EXCEPT_IPF_IA32_INTERRUPT               47
```

Description

The **RegisterExceptionCallback()** function registers and enables an exception callback function for the specified exception. The specified exception must be valid for the instruction set architecture. To unregister the callback function and stop servicing the exception, call **RegisterExceptionCallback()** passing a **NULL** *ExceptionCallback* parameter.

The implementation must handle saving and restoring the processor context to/from the system context record around calls to the registered callback function. No chaining of exception handlers is allowed.

It is the responsibility of the caller to insure all parameters are correct. There is no provision for parameter checking by **RegisterExceptionCallback()**. The implementation behavior when an invalid parameter is passed is not defined by this specification.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_ALREADY_STARTED	Non-**NULL** *ExceptionCallback* parameter when a callback function was previously registered.
EFI_OUT_OF_RESOURCES	System has insufficient memory resources to register new callback function.

EFI_DEBUG_SUPPORT_PROTOCOL.InvalidateInstructionCache()

Summary

Invalidates processor instruction cache for a memory range. Subsequent execution in this range causes a fresh memory fetch to retrieve code to be executed.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_INVALIDATE_INSTRUCTION_CACHE) (
  IN EFI_DEBUG_SUPPORT_PROTOCOL   *This,
  IN UINTN                        ProcessorIndex,
  IN VOID                         *Start,
  IN UINT64                       Length
  );
```

Parameters

This	A pointer to the **EFI_DEBUG_SUPPORT_PROTOCOL** instance. Type **EFI_DEBUG_SUPPORT_PROTOCOL** is defined in Section 16.2.
ProcessorIndex	Specifies which processor's instruction cache is to be invalidated.
Start	Specifies the physical base of the memory range to be invalidated.
Length	Specifies the minimum number of bytes in the processor's instruction cache to invalidate.

Description

Typical operation of a debugger may require modifying the code image that is under debug. This can occur for many reasons, but is typically done to insert/remove software break instructions. Some processor architectures do not have coherent instruction and data caches so modifications to the code image require that the instruction cache be explicitly invalidated in that memory region.

The **InvalidateInstructionCache()** function abstracts this operation from the debug agent and provides a general purpose capability to invalidate the processor's instruction cache.

It is the responsibility of the caller to insure all parameters are correct. There is no provision for parameter checking by **RegisterExceptionCallback()**. The implementation behavior when an invalid parameter is passed is not defined by this specification.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.

16.3 EFI Debugport Protocol

This section defines the EFI Debugport protocol. This protocol is used by debug agent to communicate with the remote debug host.

16.3.1 EFI Debugport Overview

Historically, remote debugging has typically been done using a standard UART serial port to connect the host and target. This is obviously not possible in a legacy reduced system that does not have a UART. The Debugport protocol solves this problem by providing an abstraction that can support many different types of debugport hardware. The debug agent should use this abstraction to communicate with the host.

The interface is minimal with only reset, read, and write abstractions. Since these functions are called in interrupt context, none of them may call any EFI services or other protocol interfaces.

Debugport selection and configuration is handled by setting defaults via an environment variable which contains a full device path to the debug port. This environment variable is used during the debugport driver's initialization to configure the debugport correctly. The variable contains a full device path to the debugport, with the last node (prior to the terminal node) being a debugport messaging node. See section 16.3.2 for details.

The driver must also produce an instance of the EFI Device Path protocol to indicate what hardware is being used for the debugport. This may be used by the OS to maintain the debugport across a call to **`ExitBootServices()`**.

EFI_DEBUGPORT_PROTOCOL

Summary

This protocol provides the communication link between the debug agent and the remote host.

GUID

```
#define EFI_DEBUGPORT_PROTOCOL_GUID  \
{0xEBA4E8D2,0x3858,0x41EC,0xA2,0x81,0x26,0x47,0xBA,0x96,0x60,0xD0}
```

Protocol Interface Structure

```
typedef struct {
  EFI_DEBUGPORT_RESET                   Reset;
  EFI_DEBUGPORT_WRITE                   Write;
  EFI_DEBUGPORT_READ                    Read;
  EFI_DEBUGPORT_POLL                    Poll;
} EFI_DEBUGPORT_PROTOCOL;
```

Parameters

Reset	Resets the debugport hardware.
Write	Send a buffer of characters to the debugport device.
Read	Receive a buffer of characters from the debugport device.
Poll	Determine if there is any data available to be read from the debugport device.

Description

The Debugport protocol is used for byte stream communication with a debugport device. The debugport can be a standard UART Serial port, a USB-based character device, or potentially any character-based I/O device.

The attributes for all UART-style debugport device interfaces are defined in the DEBUGPORT variable (see Section 16.3.3).

EFI_DEBUGPORT_PROTOCOL.Reset()

Summary

Resets the debugport.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DEBUGPORT_RESET) (
  IN EFI_DEBUGPORT_PROTOCOL       *This
  );
```

Parameters

This	A pointer to the **EFI_DEBUGPORT_PROTOCOL** instance. Type **EFI_DEBUGPORT_PROTOCOL** is defined in Section 16.3.

Description

The **Reset()** function resets the debugport device.

It is the responsibility of the caller to insure all parameters are valid. There is no provision for parameter checking by **Reset()**. The implementation behavior when an invalid parameter is passed is not defined by this specification.

Status Codes Returned

EFI_SUCCESS	The debugport device was reset and is in usable state.
EFI_DEVICE_ERROR	The debugport device could not be reset and is unusable.

EFI_DEBUGPORT_PROTOCOL.Write()

Summary

Writes data to the debugport.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DEBUGPORT_WRITE) (
  IN EFI_DEBUGPORT_PROTOCOL      *This,
  IN UINT32                      Timeout,
  IN OUT UINTN                   *BufferSize,
  IN VOID                        *Buffer
  );
```

Parameters

This	A pointer to the **EFI_DEBUGPORT_PROTOCOL** instance. Type **EFI_DEBUGPORT_PROTOCOL** is defined in Section 16.3.
Timeout	The number of microseconds to wait before timing out a write operation.
BufferSize	On input, the requested number of bytes of data to write. On output, the number of bytes of data actually written.
Buffer	A pointer to a buffer containing the data to write.

Description

The **Write()** function writes the specified number of bytes to a debugport device. If a timeout error occurs while data is being sent to the debugport, transmission of this buffer will terminate, and **EFI_TIMEOUT** will be returned. In all cases the number of bytes actually written to the debugport device is returned in *BufferSize*.

It is the responsibility of the caller to insure all parameters are valid. There is no provision for parameter checking by **Write()**. The implementation behavior when an invalid parameter is passed is not defined by this specification.

Status Codes Returned

EFI_SUCCESS	The data was written.
EFI_DEVICE_ERROR	The device reported an error.
EFI_TIMEOUT	The data write was stopped due to a timeout.

EFI_DEBUGPORT_PROTOCOL.Read()

Summary

Reads data from the debugport.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DEBUGPORT_READ) (
  IN EFI_DEBUGPORT_PROTOCOL        *This,
  IN UINT32                        Timeout,
  IN OUT UINTN                     *BufferSize,
  OUT VOID                         *Buffer
  );
```

Parameters

This	A pointer to the **EFI_DEBUGPORT_PROTOCOL** instance. Type **EFI_DEBUGPORT_PROTOCOL** is defined in Section 16.3.
Timeout	The number of microseconds to wait before timing out a read operation.
BufferSize	A pointer to an integer which, on input contains the requested number of bytes of data to read, and on output contains the actual number of bytes of data read and returned in *Buffer*.
Buffer	A pointer to a buffer into which the data read will be saved.

Description

The **Read()** function reads a specified number of bytes from a debugport. If a timeout error or an overrun error is detected while data is being read from the debugport, then no more characters will be read, and **EFI_TIMEOUT** will be returned. In all cases the number of bytes actually read is returned in **BufferSize*.

It is the responsibility of the caller to insure all parameters are valid. There is no provision for parameter checking by **Read()**. The implementation behavior when an invalid parameter is passed is not defined by this specification.

Status Codes Returned

EFI_SUCCESS	The data was read.
EFI_DEVICE_ERROR	The debugport device reported an error.
EFI_TIMEOUT	The operation was stopped due to a timeout or overrun.

EFI_DEBUGPORT_PROTOCOL.Poll()

Summary

Checks to see if any data is available to be read from the debugport device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DEBUGPORT_POLL) (
  IN EFI_DEBUGPORT_PROTOCOL        *This
  );
```

Parameters

This	A pointer to the **EFI_DEBUGPORT_PROTOCOL** instance. Type **EFI_DEBUGPORT_PROTOCOL** is defined in Section 16.3.

Description

The **Poll()** function checks if there is any data available to be read from the debugport device and returns the result. No data is actually removed from the input stream. This function enables simpler debugger design since buffering of reads is not necessary by the caller.

Status Codes Returned

EFI_SUCCESS	At least one byte of data is available to be read.
EFI_NOT_READY	No data is available to be read.
EFI_DEVICE_ERROR	The debugport device is not functioning correctly.

16.3.2 Debugport Device Path

The debugport driver must establish and maintain an instance of the EFI Device Path protocol for the debugport. A graceful handoff of debugport ownership between the EFI Debugport driver and an OS debugport driver requires that the OS debugport driver can determine the type, location, and configuration of the debugport device.

The Debugport Device Path is a vendor-defined messaging device path with no data, only a GUID. It is used at the end of a conventional device path to tag the device for use as the debugport. For example, a typical UART debugport would have the following fully qualified device path:

ACPI(PciRootBridge)/Pci(0x1f,0)/ACPI(PNP0501,0)/UART(115200,n,8,1)/DebugPort()

The Vendor_GUID that defines the debugport device path is the same as the debugport protocol GUID, as defined below.

```
#define DEVICE_PATH_MESSAGING_DEBUGPORT     \
            EFI_DEBUGPORT_PROTOCOL_GUID
```

Table 16-1 shows all fields of the debugport device path.

Table 16-1. Debugport Messaging Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path.
Sub Type	1	1	Sub Type 10 – Vendor.
Length	2	2	Length of this structure in bytes. Length is 20 bytes.
Vendor_GUID	4	16	**DEVICE_PATH_MESSAGING_DEBUGPORT**.

16.3.3 EFI Debugport Variable

Even though there may be more than one hardware device that could function as a debugport in a system, only one debugport may be active at a time. The DEBUGPORT variable is used to declare which hardware device will act as the debugport, and what communication parameters it should assume.

Like all EFI variables, the DEBUGPORT variable has both a name and a GUID. The name is "DEBUGPORT." The GUID is the same as the **EFI_DEBUGPORT_PROTOCOL_GUID**:

```
#define EFI_DEBUGPORT_VARIABLE_NAME L"DEBUGPORT"
#define EFI_DEBUGPORT_VARIABLE_GUID EFI_DEBUGPORT_PROTOCOL_GUID
```

The data contained by the DEBUGPORT variable is a fully qualified debugport device path (see Section 16.3.2).

The desired communication parameters for the debugport are declared in the DEBUGPORT variable. The debugport driver must read this variable during initialization to determine how to configure the debug port.

To reduce the required complexity of the debugport driver, the debugport driver is not required to support all possible combinations of communication parameters. What combinations of parameters are possible is implementation specific.

Additionally debugport drivers implemented for PNP0501 devices, that is debugport devices with a PNP0501 ACPI node in the device path, must support the following defaults. These defaults must be used in the absence of a DEBUGPORT variable, or when the communication parameters specified in the DEBUGPORT variable are not supported by the driver.

- Baud : 115200
- 8 data bits
- No parity
- 1 stop bit
- No flow control (See Appendix A for flow control details)

In the absence of the DEBUGPORT variable, the selection of which port to use as the debug port is implementation specific.

Future revisions of this specification may define new defaults for other debugport types.

The debugport device path must be constructed to reflect the actual settings for the debugport. Any code needing to know the state of the debug port must reference the device path rather than the DEBUGPORT variable, since the debugport may have assumed a default setting in spite of the existence of the DEBUGPORT variable.

If it is not possible to configure the debug port using either the settings declared in the DEBUGPORT variable or the default settings for the particular debugport type, the driver initialization must not install any protocol interfaces and must exit with an error.

16.4 EFI Debug Support Table

This chapter defines the EFI Debug Support Table which is used by the debug agent or an external debugger to determine loaded image information in a quiescent manner.

16.4.1 Overview

Every executable image loaded in EFI is represented by an EFI handle populated with an instance of the **`LOADED_IMAGE`** protocol. This handle is known as an "image handle." The associated Loaded Image protocol provides image information that is of interest to a source level debugger. Normal EFI executables can access this information by using EFI services to locate all instances of the Loaded Image protocol.

A debugger has two problems with this scenario. First, if it is an external hardware debugger, the location of the EFI system table is not known. Second, even if the location of the EFI system table is known, the services contained therein are generally unavailable to a debugger either because it is an on-target debugger that is running in interrupt context, or in the case of an external hardware debugger there is no debugger code running on the target at all.

Since a source level debugger must be capable of determining image information for all loaded images, an alternate mechanism that does not use EFI services must be provided. Two features are added to the EFI system software to enable this capability.

First, an alternate mechanism of locating the EFI system table is required. A check-summed structure containing the physical address of the EFI system table is created and located on a 4M aligned memory address. A hardware debugger can search memory for this structure to determine the location of the EFI system table.

Second, an **`EFI_CONFIGURATION_TABLE`** is published that leads to a database of pointers to all instances of the Loaded Image protocol. Several layers of indirection are used to allow dynamically managing the data as images are loaded and unloaded. Once the address of the EFI system table is known, it is possible to discover a complete and accurate list of EFI images. (Note that the EFI core itself must be represented by an instance of the Loaded Image protocol.)

Figure 16-1 illustrates the table indirection and pointer usage.

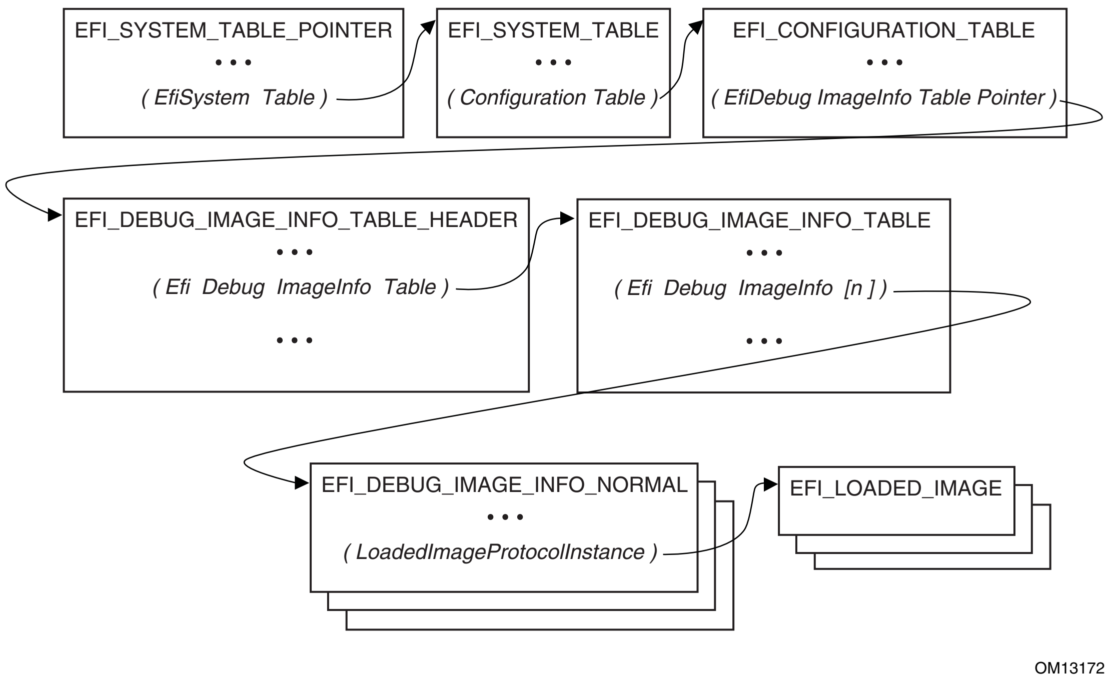

EFI_SYSTEM_TABLE_POINTER
• • •
(EfiSystem Table)
EFI_SYSTEM_TABLE
• • •
(Configuration Table)
EFI_CONFIGURATION_TABLE
• • •
(EfiDebug ImageInfo Table Pointer)
EFI_DEBUG_IMAGE_INFO_TABLE_HEADER
• • •
(Efi Debug ImageInfo Table)
• • •
EFI_DEBUG_IMAGE_INFO_TABLE
• • •
(Efi Debug ImageInfo [n])
• • •
EFI_DEBUG_IMAGE_INFO_NORMAL
• • •
(LoadedImageProtocolInstance)
EFI_LOADED_IMAGE
OM13172

Figure 16-1. Debug Support Table Indirection and Pointer Usage

16.4.2 EFI System Table Location

The EFI system table can be located by an off-target hardware debugger by searching for the **EFI_SYSTEM_TABLE_POINTER** structure. The **EFI_SYSTEM_TABLE_POINTER** structure is located on a 4M boundary as close to the top of physical memory as feasible. It may be found searching for **the EFI_SYSTEM_TABLE_SIGNATURE** on each 4M boundary starting at the top of memory and scanning down. When the signature is found, the entire structure must verified using the *Crc32* field. The 32-bit CRC of the entire structure is calculated assuming the *Crc32* field is zero. This value is then written to the *Crc32* field.

```
typedef struct _EFI_SYSTEM_TABLE_POINTER {
  UINT64                 Signature;
  EFI_PHYSICAL_ADDRESS   EfiSystemTableBase;
  UINT32                 Crc32;
} EFI_SYSTEM_TABLE_POINTER;
```

Signature	A constant **UINT64** that has the value **EFI_SYSTEM_TABLE_SIGNATURE** (see the EFI 1.0 specification).
EfiSystemTableBase	The physical address of the EFI system table.
Crc32	A 32-bit CRC value that is used to verify the **EFI_SYSTEM_TABLE_POINTER** structure is valid.

16.4.3 EFI Image Info

The **EFI_DEBUG_IMAGE_INFO_TABLE** is an array of pointers to **EFI_DEBUG_IMAGE_INFO** unions. Each member of an **EFI_DEBUG_IMAGE_INFO** union is a pointer to a data structure representing a particular image type. For each image that has been loaded, there is an appropriate image data structure with a pointer to it stored in the **EFI_DEBUG_IMAGE_INFO_TABLE**. Data structures for normal images and SMM images are defined. All other image types are reserved for future use.

The process of locating the **EFI_DEBUG_IMAGE_INFO_TABLE** begins with an EFI configuration table.

```
  //
  // EFI_DEBUG_IMAGE_INFO_TABLE configuration table
  //     GUID declaration - {49152E77-1ADA-4764-B7A2-7AFEFED95E8B}
  //
  #define EFI_DEBUG_IMAGE_INFO_TABLE_GUID    \
{ 0x49152E77,0x1ADA,0x4764,0xB7,0xA2,0x7A,0xFE,0xFE,0xD9,0x5E,0x8B }
```

The configuration table leads to an **EFI_DEBUG_IMAGE_INFO_TABLE_HEADER** structure that contains a pointer to the **EFI_DEBUG_IMAGE_INFO_TABLE** and some status bits that are used to control access to the **EFI_DEBUG_IMAGE_INFO_TABLE** when it is being updated.

```
//
// UpdateStatus bits
//
#define EFI_DEBUG_IMAGE_INFO_UPDATE_IN_PROGRESS  0x01
#define EFI_DEBUG_IMAGE_INFO_TABLE_MODIFIED                0x02

typedef struct {
  volatile UINT32         UpdateStatus;
  UINT32                  TableSize;
  EFI_DEBUG_IMAGE_INFO    *EfiDebugImageInfoTable;
} EFI_DEBUG_IMAGE_INFO_TABLE_HEADER;
```

UpdateStatus	*UpdateStatus* is used by the system to indicate the state of the debug image info table. The **EFI_DEBUG_IMAGE_INFO_UPDATE_IN_PROGRESS** bit must be set when the table is being modified. Software consuming the table must qualify the access to the table with this bit. The **EFI_DEBUG_IMAGE_INFO_TABLE_MODIFIED** bit is always set by software that modifies the table. It may be cleared by software that consumes the table once the entire table has been read. It is essentially a sticky version of the **EFI_DEBUG_IMAGE_INFO_UPDATE_IN_PROGRESS** bit and is intended to provide an efficient mechanism to minimize the number of times the table must be scanned by the consumer.
TableSize	The number of **EFI_DEBUG_IMAGE_INFO** elements in the array pointed to by *EfiDebugImageInfoTable*.
EfiDebugImageInfoTable	A pointer to the first element of an array of **EFI_DEBUG_IMAGE_INFO** structures.

```
#define EFI_DEBUG_IMAGE_INFO_TYPE_NORMAL  0x01

typdef union {
  UINT32                       *ImageInfoType;
  EFI_DEBUG_IMAGE_INFO_NORMAL  *NormalImage;
} EFI_DEBUG_IMAGE_INFO;

typedef struct {
  UINT32                       ImageInfoType;
  EFI_LOADED_IMAGE_PROTOCOL    *LoadedImageProtocolInstance;
  EFI_HANDLE                   ImageHandle;
} EFI_DEBUG_IMAGE_INFO_NORMAL;
```

ImageInfoType	Indicates the type of image info structure. For PE32 EFI images, this is set to **EFI_DEBUG_IMAGE_INFO_TYPE_NORMAL**.
LoadedImageProtocolInstance	A pointer to an instance of the loaded image protocol for the associated image.
ImageHandle	Indicates the image handle of the associated image.

17 Protocols - Compression Algorithm Specification

In EFI firmware storage, binary codes/data are often compressed to save storage space. These compressed codes/data are extracted into memory for execution at boot time. This demands an efficient lossless compression/decompression algorithm. The compressor must produce small compressed images, and the decompressor must operate fast enough to avoid delays at boot time.

This chapter describes in detail the EFI compression/decompression algorithm, as well as the EFI Decompress Protocol. The EFI Decompress Protocol provides a standard decompression interface for use at boot time.

17.1 Algorithm Overview

In this chapter the term "**character**" denotes a single byte and the term "**string**" denotes a series of concatenated characters.

The compression/decompression algorithm used in EFI firmware storage is a combination of the LZ77 algorithm and Huffman Coding. The LZ77 algorithm replaces a repeated string with a pointer to the previous occurrence of the string. Huffman Coding encodes symbols in a way that the more frequently a symbol appears in a text, the shorter the code that is assigned to it.

The compression process contains two steps:

- The first step is to find repeated strings (using LZ77 algorithm) and produce intermediate data. Beginning with the first character, the compressor scans the source data and determines if the characters starting at the current position can form a string previously appearing in the text. If a long enough matching string is found, the compressor will output a pointer to the string. If the pointer occupies more space than the string itself, the compressor will output the original character at the current position in the source data. Then the compressor advances to the next position and repeats the process. To speed up the compression process, the compressor dynamically maintains a **String Info Log** to record the positions and lengths of strings encountered, so that string comparisons are performed quickly by looking up the String Info Log.
  Because a compressor cannot have unlimited resources, as the compression continues the compressor removes "old" string information. This prevents the String Info Log from becoming too large. As a result, the algorithm can only look up repeated strings within the range of a fixed-sized "sliding window" behind the current position.
  In this way, a stream of intermediate data is produced which contains two types of symbols: the **Original Characters** (to be preserved in the decompressed data), and the **Pointers** (representing a previous string). A Pointer consists of two elements: the **String Position** and the **String Length**, representing the location and the length of the target string, respectively.

- To improve the compression ratio further, Huffman Coding is utilized as the second step.

  The intermediate data (consisting of original characters and pointers) is divided into **Blocks** so that the compressor can perform Huffman Coding on a Block immediately after it is generated; eliminating the need for a second pass from the beginning after the intermediate data has been generated. Also, since symbol frequency distribution may differ in different parts of the intermediate data, Huffman Coding can be optimized for each specific Block. The compressor determines Block Size for each Block according to the specifications defined in section 17.2, "Data Format."

  In each Block, two symbol sets are defined for Huffman Coding. The **Char&Len Set** consists of the Original Characters plus the String Lengths and the **Position Set** consists of String Positions (Note that the two elements of a Pointer belong to separate symbol sets). The Huffman Coding schemes applied on these two symbol sets are independent.

  The algorithm uses "canonical" Huffman Coding so a Huffman tree can be represented as an array of code lengths in the order of the symbols in the symbol set. This code length array represents the Huffman Coding scheme for the symbol set. Both the Char&Len Set code length array and the Position Set code length array appear in the Block Header.

  Huffman coding is used on the code length array of the Char&Len Set to define a third symbol set. The **Extra Set** is defined based on the code length values in the Char&Len Set code length array. The code length array for the Huffman Coding of Extra Set also appears in the Block Header together with the other two code length arrays. For exact format of the Block Header, see section 17.2.3.1, "Block Header."

The decompression process is straightforward given that the compression process is known. The decompressor scans the compressed data and decodes the symbols one by one, according to the Huffman code mapping tables generated from code length arrays. Along the process, if it encounters an original character, it outputs it; if it encounters a pointer, it looks it up in the already decompressed data and outputs the associated string.

17.2 Data Format

This section describes in detail the format of the compressed data produced by the compressor. The compressed data serves as input to the decompressor and can be fully extracted to the original source data.

17.2.1 Bit Order

In computer data representation, a byte is the minimum unit and there is no differentiation in the order of bits within a byte. However, the compressed data is a sequence of bits rather than a sequence of bytes and as a result the order of bits in a byte needs to be defined. In a compressed data stream, the higher bits are defined to precede the lower bits in a byte. Figure 17-1 illustrates a compressed data sequence written as bytes from left to right. For each byte, the bits are written in an order with bit 7 (the highest bit) at the left and bit 0 (the lowest bit) at the right. Concatenating the bytes from left to right forms a bit sequence.

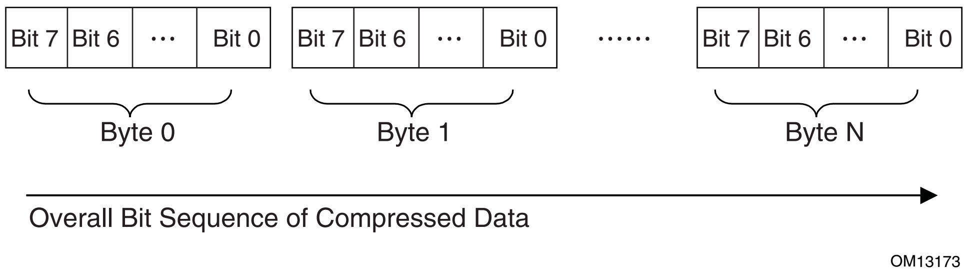


Figure 17-1. Bit Sequence of Compressed Data

The bits of the compressed data are actually formed by a sequence of data units. These data units have variable bit lengths. The bits of each data unit are arranged so that the higher bit of the data unit precedes the lower bit of the data unit.

17.2.2 Overall Structure

The compressed data begins with two 32-bit numerical fields: the compressed size and the original size. The compressed data following these two fields is composed of one or more Blocks. Each Block is a unit for Huffman Coding with a coding scheme independent of the other Blocks. Each Block is composed of a Block Header containing the Huffman code trees for this Block and a Block Body with the data encoded using the coding scheme defined by the Huffman trees. The compressed data is terminated by an additional byte of zero.

The overall structure of the compressed data is shown in Figure 17-2.

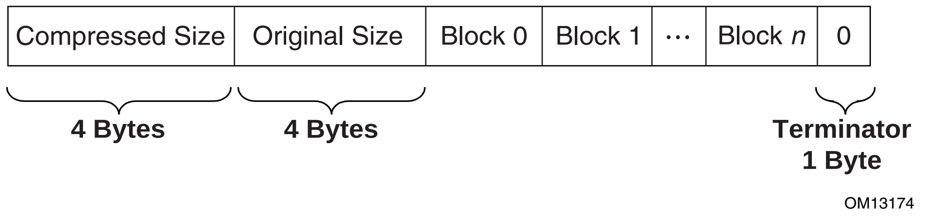


Figure 17-2. Compressed Data Structure

Note the following:

- Blocks are of variable lengths.
- Block lengths are counted by bits and not necessarily divisible by 8. Blocks are tightly packed (there are no padding bits between blocks). Neither the starting position nor ending position of a Block is necessarily at a byte boundary. However, if the last Block is not terminated at a byte boundary, there should be some bits of 0 to fill up the remaining bits of the last byte of the block, before the terminator byte of 0.
- Compressed Size =
  Size in bytes of (Block 0 + Block 1 + … + Block N + Filling Bits (if any) + Terminator).
- Original Size is the size in bytes of original data.
- Both Compressed Size and Original Size are "little endian" (starting from the least significant byte).

17.2.3 Block Structure

A Block is composed of a Block Header and a Block Body, as shown in Figure 17-3. These two parts are packed tightly (there are no padding bits between them). The lengths in bits of Block Header and Block Body are not necessarily divisible by eight.

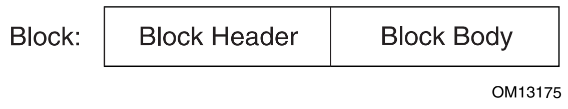


Figure 17-3. Block Structure

17.2.3.1 Block Header

The Block Header contains the Huffman encoding information for this block. Since "canonical" Huffman Coding is being used, a Huffman tree is represented as an array of code lengths in increasing order of the symbols in the symbol set. Code lengths are limited to be less than or equal to 16 bits. This requires some extra handling of Huffman codes in the compressor, which is described in section 17.3, "Compressor Design."

There are three code length arrays for three different symbol sets in the Block Header: one for the Extra Set, one for the Char&Len Set, and one for the Position Set.

The Block Header is composed of the tightly packed (no padding bits) fields described in Table 17-1.

Table 17-1. Block Header Fields

Field Name	Length (bits)	Description
Block Size	16	The size of this Block. Block Size is defined as the number of original characters plus the number of pointers that appear in the Block Body: Block Size = Number of Original Characters in the Block Body + Number of Pointers in the Block Body.
Extra Set Code Length Array Size	5	The number of code lengths in the Extra Set Code Length Array. The Extra Set Code Length Array contains code lengths of the Extra Set in increasing order of the symbols, and if all symbols greater than a certain symbol have zero code length, the Extra Set Code Length Array terminates at the last nonzero code length symbol. Since there are 19 symbols in the Extra Set (see the description of the Char&Len Set Code Length Array), the maximum Extra Set Code Length Array Size is 19.
Extra Set Code Length Array	Variable	If Extra Set Code Length Array Size is 0, then this field is a 5-bit value that represents the only Huffman code used. If Extra Set Code Length Array Size is not 0, then this field is an encoded form of a concatenation of code lengths in increasing order of the symbols. The concatenation of Code lengths are encoded as follows: If a code length is less than 7, then it is encoded as a 3-bit value; If a code length is equal to or greater than 7, then it is encoded as a series of "1"s followed by a terminating "0." The number of "1"s = Code length – 4. For example, code length "ten" is encoded as "1111110"; code length "seven" is encoded as "1110." After the third length of the code length concatenation, a 2-bit value is used to indicate the number of consecutive zero lengths immediately after the third length. (Note this 2-bit value only appears once after the third length, and does NOT appear multiple times after every 3rd length.) This 2-bit value ranges from 0 to 3. For example, if the 2-bit value is "00," then it means there are no zero lengths at the point, and following encoding starts from the fourth code length; if the 2-bit value is "10" then it means the fourth and fifth length are zero and following encoding starts from the sixth code length.
Position Set Code Length Array Size	4	The number of code lengths in the Position Set Code Length Array. The Position Set Code Length Array contains code lengths of Position Set in increasing order of the symbols in the Position Set, and if all symbols greater than a certain symbol have zero code length, the Position Set Code Length Array terminates at the last nonzero code length symbol. Since there are 14 symbols in the Position Set (see 3.3.2), the maximum Position Set Code Length Array Size is 14.

continued

Table 17-1. Block Header Fields (continued)

Field Name	Length (bits)	Description
Char&Len Set Code Length Array	Variable	If Char&Len Set Code Length Array Size is 0, then this field is a 9-bit value that represents the only Huffman code used. If Char&Len Set Code Length Array Size is not 0, then this field is an encoded form of a concatenation of code lengths in increasing order of the symbols. The concatenation of Code lengths are two-step encoded: Step 1: If a code length is not zero, then it is encoded as "code length + 2"; If a code length is zero, then the number of consecutive zero lengths starting from this code length is counted -- If the count is equal to or less than 2, then the code "0" is used for each zero length; if the count is greater than 2 and less than 19, then the code "1" followed by a 4-bit value of "count – 3" is used for these consecutive zero lengths; if the count is equal to 19, then it is treated as "1 + 18," and a code "0" and a code "1" followed by a 4-bit value of "15" are used for these consecutive zero lengths; if the count is greater than 19, then the code "2" followed by a 9-bit value of "count – 20" is used for these consecutive zero lengths. Step 2: The second step encoding is a Huffman encoding of the codes produced by first step. (While encoding codes "1" and "2," their appended values are not encoded and preserved in the resulting text). The code lengths of generated Huffman tree are just the contents of the Extra Set Code Length Array.
Position Set Code Length Array Size	4	The number of code lengths in the Position Set Code Length Array. The Position Set Code Length Array contains code lengths of Position Set in increasing order of the symbols in the Position Set, and if all symbols greater than a certain symbol have zero code length, the Position Set Code Length Array terminates at the last nonzero code length symbol. Since there are 14 symbols in the Position Set (see 3.3.2), the maximum Position Set Code Length Array Size is 14.
Position Set Code Length Array	Variable	If Position Set Code Length Array Size is 0, then this field is a 5-bit value that represents the only Huffman code used. If Position Set Code Length Array Size is not 0, then this field is an encoded form of a concatenation of code lengths in increasing order of the symbols. The concatenation of Code lengths are encoded as follows: If a code length is less than 7, then it is encoded as a normal 3-bit value; If a code length is equal to or greater than 7, then it is encoded as a series of "1"s followed by a terminating "0." The number of "1"s = Code length – 4. For example, code length "10" is encoded as "1111110"; code length "7" is encoded as "1110."

17.2.3.2 Block Body

The Block Body is simply a mixture of Original Characters and Pointers, while each Pointer has two elements: String Length preceding String Position. All these data units are tightly packed together.

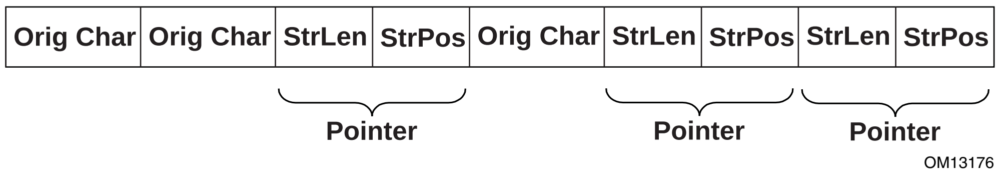


Figure 17-4. Block Body

The Original Characters, String Lengths and String Positions are all Huffman coded using the Huffman trees presented in the Block Header, with some additional variations. The exact format is described below:

An Original Character is a byte in the source data. A String Length is a value that is greater than 3 and less than 257 (this range should be ensured by the compressor). By calculating "(String Length – 3) | 0x100," a value set is obtained that ranges from 256 to 509. By combining this value set with the value set of Original Characters (0 ~ 255), the Char&Len Set (ranging from 0 to 509) is generated for Huffman Coding.

A String Position is a value that indicates the distance between the current position and the target string. The String Position value is defined as "Current Position – Starting Position of the target string - 1." The String Position value ranges from 0 to 8190 (so 8192 is the "sliding window" size, and this range should be ensured by the compressor). The lengths of the String Position values (in binary form) form a value set ranging from 0 to 13 (it is assumed that value 0 has length of 0). This value set is the Position Set for Huffman Coding. The full representation of a String Position value is composed of two consecutive parts: one is the Huffman code for the value length; the other is the actual String Position value of "length - 1" bits (excluding the highest bit since the highest bit is always "1"). For example, String Position value 18 is represented as: Huffman code for "5" followed by "0010." If the value length is 0 or 1, then no value is appended to the Huffman code. This kind of representation favors small String Position values, which is a hint for compressor design.

17.3 Compressor Design

The compressor takes the source data as input and produces a compressed image. This section describes the design used in one possible implementation of a compressor that follows the EFI 1.10 Compression Algorithm. The source code that illustrates an implementation of this specific design is listed in Appendix H.

17.3.1 Overall Process

The compressor scans the source data from the beginning, character by character. As the scanning proceeds, the compressor generates Original Characters or Pointers and outputs the compressed data packed in a series of Blocks representing individual Huffman coding units.

The compressor maintains a String Info Log containing data that facilitates string comparison. Old data items are deleted and new data items are inserted regularly.

The compressor does not output a Pointer immediately after it sees a matching string for the current position. Instead, it delays its decision until it gets the matching string for the next position. The compressor has two criteria at hand: one is that the former match length should be no shorter than three characters; the other is that the former match length should be no shorter than the latter match length. Only when these two criteria are met does the compressor output a Pointer to the former matching string.

The overall process of compression can be described by following pseudo code:

```
Set the Current Position at the beginning of the source data;
Delete the outdated string info from the String Info Log;
Search the String Info Log for matching string;
Add the string info of the current position into the String Info Log;
WHILE not end of source data DO
  Remember the last match;
  Advance the Current Position by 1;
  Delete the outdated String Info from the String Info Log;
  Search the String Info Log for matching string;
  Add the string info of the Current Position into the String Info Log;
  IF the last match is shorter than 3 characters or this match is longer than
  the last match THEN
    Call Output()* to output the character at the previous position as an
    Original Character;
  ELSE
    Call Output()* to output a Pointer to the last matching string;
    WHILE (--last match length) > 0 DO
      Advance the Current Position by 1;
      Delete the outdated piece of string info from the String Info Log;
      Add the string info of the current position into the String Info Log;
    ENDWHILE
  ENDIF
ENDWHILE
```

The *Output()* is the function that is responsible for generating Huffman codes and Blocks. It accepts an Original Character or a Pointer as input and maintains a Block Buffer to temporarily store data units that are to be Huffman coded. The following pseudo code describes the function:

```
FUNCTION NAME: Output
INPUT: an Original Character or a Pointer

Put the Original Character or the Pointer into the Block Buffer;
Advance the Block Buffer position pointer by 1;
IF the Block Buffer is full THEN
  Encode the Char&Len Set in the Block buffer;
  Encode the Position Set in the Block buffer;
  Encode the Extra Set;
  Output the Block Header containing the code length arrays;
  Output the Block Body containing the Huffman encoded Original Characters and
  Pointers;
  Reset the Block Buffer position pointer to point to the beginning of the
  Block buffer;
ENDIF
```

17.3.2 String Info Log

The provision of the String Info Log is to speed up the process of finding matching strings. The design of this has significant impact on the overall performance of the compressor. This section describes in detail how String Info Log is implemented and the typical operations on it.

17.3.2.1 Data Structures

The String Info Log is implemented as a set of search trees. These search trees are dynamically updated as the compression proceeds through the source data. The structure of a typical search tree is depicted in Figure 17-5.

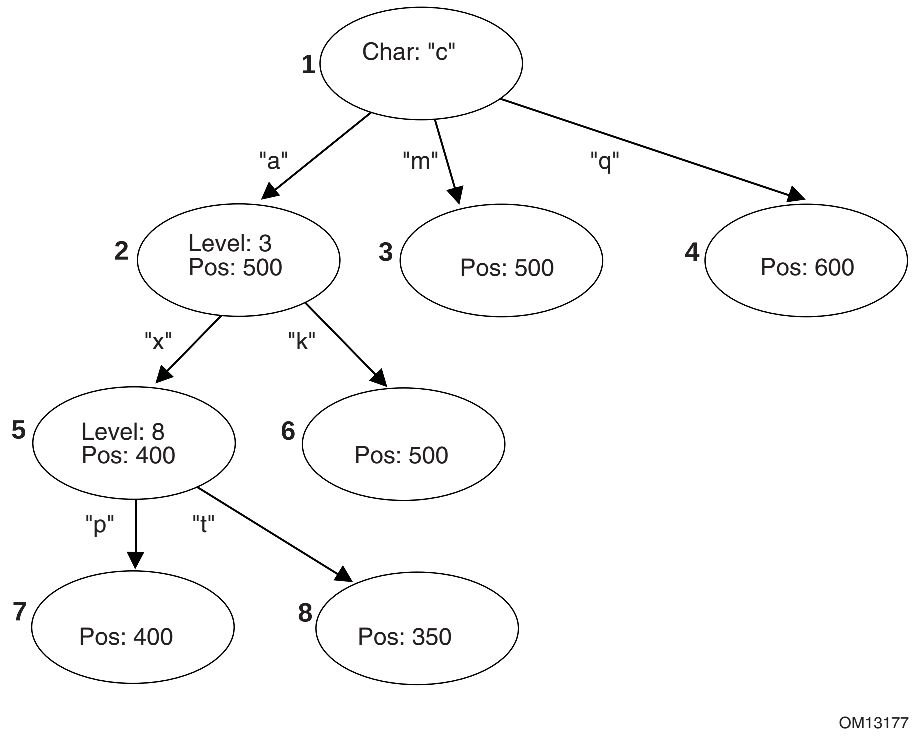


Figure 17-5. String Info Log Search Tree

There are three types of nodes in a search tree: the root node, internal nodes, and leaves. The root node has a “character” attribute, which represents the starting character of a string. Each edge also has a “character” attribute, which represents the next character in the string. Each internal node has a “level” attribute, which indicates the character on any edge that leads to its child nodes is the “level + 1”th character in the string. Each internal node or leaf has a “position” attribute that indicates the string’s starting position in the source data.

To speed up the tree searching, a hash function is used. Given the parent node and the edge-character, the hash function will quickly find the expected child node.

17.3.2.2 Searching the Tree

Traversing the search tree is performed as follows:

The following example uses the search tree shown in Figure 17-5 above. Assume that the current position in the source data contains the string "camxrsxpj…."

1. The starting character "c" is used to find the root of the tree. The next character "a" is used to follow the edge from node 1 to node 2. The "position" of node 2 is 500, so a string starting with "ca" can be found at position 500. The string at the current position is compared with the string starting at position 500.
2. Node 2 is at Level 3; so at most three characters are compared. Assume that the three-character comparison passes.
3. The fourth character "x" is used to follow the edge from Node 2 to Node 5. The position value of node 5 is 400, which means there is a string located in position 400 that starts with "cam" and the character at position 403 is an "x."
4. Node 5 is at Level 8, so the fifth to eighth characters of the source data are compared with the string starting at position 404. Assume the strings match.
5. At this point, the ninth character "p" has been reached. It is used to follow the edge from Node 5 to Node 7.
6. This process continues until a mismatch occurs, or the length of the matching strings exceeds the predefined MAX_MATCH_LENGTH. The most recent matching string (which is also the longest) is the desired matching string.

17.3.2.3 Adding String Info

String info needs to be added to the String Info Log for each position in the source data. Each time a search for a matching string is performed, the new string info is inserted for the current position. There are several cases that can be discussed:

1. No root is found for the first character. A new tree is created with the root node labeled with the starting character and a child leaf node with its edge to the root node labeled with the second character in the string. The "position" value of the child node is set to the current position.
2. One root node matches the first character, but the second character does not match any edge extending from the root node. A new child leaf node is created with its edge labeled with the second character. The "position" value of the new leaf child node is set to the current position.
3. A string comparison succeeds with an internal node, but a matching edge for the next character does not exist. This is similar to (2) above. A new child leaf node is created with its edge labeled with the character that does not exist. The "position" value of the new leaf child node is set to the current position.
4. A string comparison exceeds MAX_MATCH_LENGTH. Note: This only happens with leaf nodes. For this case, the "position" value in the leaf node is updated with the current position.

5. If a string comparison with an internal node or leaf node fails (mismatch occurs before the "Level + 1"th character is reached or MAX_MATCH_LENGTH is exceeded), then a "split" operation is performed as follows:

   Suppose a comparison is being performed with a level 9 Node, at position 350, and the current position is 1005. If the sixth character at position 350 is an "x" and the sixth character at position 1005 is a "y," then a mismatch will occur. In this case, a new internal node and a new child node are inserted into the tree, as depicted in Figure 17-6.

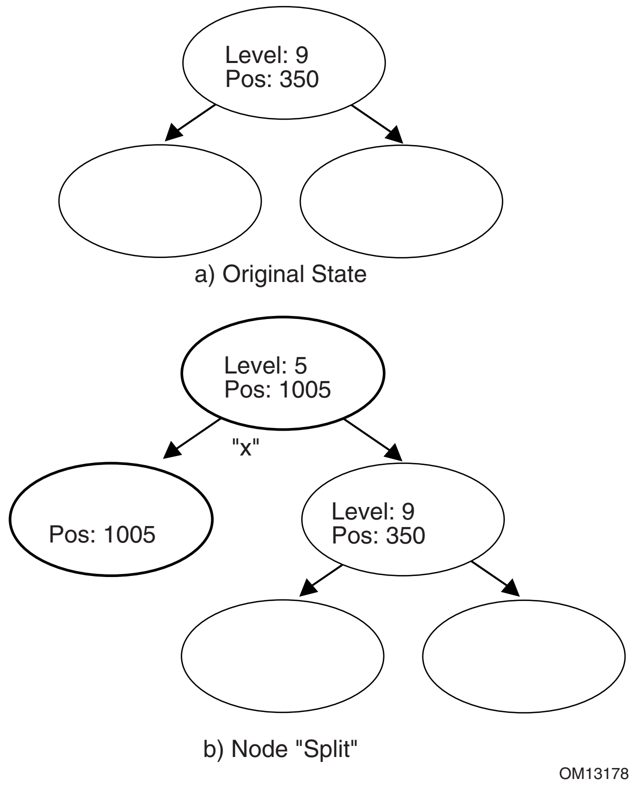


Figure 17-6. Node Split

The b) portion of Figure 17-6 has two new inserted nodes, which reflects the new string information that was found at the current position. The process splits the old node into two child nodes, and that is why this operation is called a "split."

17.3.2.4 Deleting String Info

The String Info Log will grow as more and more string information is logged. The size of the String Info Log must be limited, so outdated information must be removed on a regular basis.
A sliding window is maintained behind the current position, and the searches are always limited within the range of the sliding window. Each time the current position is advanced, outdated string information that falls outside the sliding window should be removed from the tree. The search for outdated string information is simplified by always updating the nodes' "position" attribute when searching for matching strings.

17.3.3 Huffman Code Generation

Another major component of the compressor design is generation of the Huffman Code.

Huffman Coding is applied to the Char&Len Set, the Position Set, and the Extra Set. The Huffman Coding used here has the following features:

1. The Huffman tree is represented as an array of code lengths ("canonical" Huffman Coding);
2. The maximum code length is limited to 16 bits.

The Huffman code generation process can be divided into three steps. These are the generation of Huffman tree, the adjustment of code lengths, and the code generation.

17.3.3.1 Huffman Tree Generation

This process generates a typical Huffman tree. First, the frequency of each symbol is counted, and a list of nodes is generated with each node containing a symbol and the symbol's frequency. The two nodes with the lowest frequency values are merged into a single node. This new node becomes the parent node of the two nodes that are merged. The frequency value of this new parent node is the sum of the two child nodes' frequency values. The node list is updated to include the new parent node but exclude the two child nodes that are merged. This process is repeated until there is a single node remaining that is the root of the generated tree.

17.3.3.2 Code Length Adjustment

The leaf nodes of the tree generated by the previous step represent all the symbols that were generated. Traditionally the code for each symbol is found by traversing the tree from the root node to the leaf node. Going down a left edge generates a "0," and going down a right edge generates a "1." However, a different approach is used here. The number of codes of each code length is counted. This generates a 16-element *LengthCount* array, with *LengthCount[i]* = Number Of Codes whose Code Length is *i*. Since a code length may be longer than 16 bits, the sixteenth entry of the *LengthCount* array is set to the Number Of Codes whose Code Length is greater than or equal to 16.

The *LengthCount* array goes through further adjustment described by following code:

```
INT32 i, k;
UINT32 cum;

cum = 0;
for (i = 16; i > 0; i--) {
  cum += LengthCount[i] << (16 - i);
}
while (cum != (1U << 16)) {
  LengthCount[16]--;
  for (i = 15; i > 0; i--) {
    if (LengthCount[i] != 0) {
      LengthCount[i]--;
      LengthCount[i+1] += 2;
      break;
    }
  }
  cum--;
}
```

17.3.3.3 Code Generation

In the previous step, the count of each length was obtained. Now, each symbol is going to be assigned a code. First, the length of the code for each symbol is determined. Naturally, the code lengths are assigned in such a way that shorter codes are assigned to more frequently appearing symbols. A *CodeLength* array is generated with *CodeLength[i]* = the code length of symbol *i*. Given this array, a code is assigned to each symbol using the algorithm described by the pseudo code below (the resulting codes are stored in array *Code* such that *Code[i]* = the code assigned to symbol *i*):

```
INT32    i;
UINT16   Start[18];

Start[1] = 0;

for (i = 1; i <= 16; i++) {
  Start[i + 1] = (UINT16)((Start[i] + LengthCount[i]) << 1);
}

for (i = 0; i < NumberOfSymbols; i++) {
  Code[i] = Start[CodeLength[i]]++;
}
```

The code length adjustment process ensures that no code longer than the designated length will be generated. As long as the decompressor has the *CodeLength* array at hand, it can regenerate the codes.

17.4 Decompressor Design

The decompressor takes the compressed data as input and produces the original source data. The main tasks for the decompressor are decoding Huffman codes and restoring Pointers to the strings to which they point.

The following pseudo code describes the algorithm used in the design of a decompressor. The source code that illustrates an implementation of this design is listed in Appendix I.

```
WHILE not end of data DO
  IF at block boundary THEN
    Read in the Extra Set Code Length Array;
    Generate the Huffman code mapping table for the Extra Set;
    Read in and decode the Char&Len Set Code Length Array;
    Generate the Huffman code mapping table for the Char&Len Set;
    Read in the Position Set Code Length Array;
    Generate the Huffman code mapping table for the Position Set;
  ENDIF
  Get next code;
  Look the code up in the Char&Len Set code mapping table.
  Store the result as C;
  IF C < 256 (it represents an Original Character) THEN
    Output this character;
  ELSE (it represents a String Length)
    Transform C to be the actual String Length value;
    Get next code and look it up in the Position Set code mapping table, and
    with some additional transformation, store the result as P;
    Output C characters starting from the position "Current Position - P";
  ENDIF
ENDWHILE
```

17.5 Decompress Protocol

This section provides a detailed description of the **EFI_DECOMPRESS_PROTOCOL**.

EFI_DECOMPRESS_PROTOCOL

Summary

Provides a decompression service.

GUID

```
#define EFI_DECOMPRESS_PROTOCOL_GUID  \
  {0xd8117cfe,0x94a6,0x11d4,0x9a,0x3a,0x0,0x90,0x27,0x3f,0xc1,0x4d}
```

Protocol Interface Structure

```
typedef struct _EFI_DECOMPRESS_PROTOCOL {
  EFI_DECOMPRESS_GET_INFO     GetInfo;
  EFI_DECOMPRESS_DECOMPRESS   Decompress;
} EFI_DECOMPRESS_PROTOCOL;
```

Parameters

GetInfo	Given the compressed source buffer, this function retrieves the size of the uncompressed destination buffer and the size of the scratch buffer required to perform the decompression. It is the caller's responsibility to allocate the destination buffer and the scratch buffer prior to calling **Decompress()**. See the **GetInfo()** function description.
Decompress	Decompresses a compressed source buffer into an uncompressed destination buffer. It is the caller's responsibility to allocate the destination buffer and a scratch buffer prior to making this call. See the **Decompress()** function description.

Description

The **EFI_DECOMPRESS_PROTOCOL** provides a decompression service that allows a compressed source buffer in memory to be decompressed into a destination buffer in memory. It also requires a temporary scratch buffer to perform the decompression. The **GetInfo()** function retrieves the size of the destination buffer and the size of the scratch buffer that the caller is required to allocate. The **Decompress()** function performs the decompression. The scratch buffer can be freed after the decompression is complete.

EFI_DECOMPRESS_PROTOCOL.GetInfo()

Summary

Given a compressed source buffer, this function retrieves the size of the uncompressed buffer and the size of the scratch buffer required to decompress the compressed source buffer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DECOMPRESS_GET_INFO) (
  IN    EFI_DECOMPRESS_PROTOCOL   *This,
  IN    VOID                      *Source,
  IN    UINT32                    SourceSize,
  OUT   UINT32                    *DestinationSize,
  OUT   UINT32                    *ScratchSize
  );
```

Parameters

This	A pointer to the **EFI_DECOMPRESS_PROTOCOL** instance. Type **EFI_DECOMPRESS_PROTOCOL** is defined in Section 17.5.
Source	The source buffer containing the compressed data.
SourceSize	The size, in bytes, of the source buffer.
DestinationSize	A pointer to the size, in bytes, of the uncompressed buffer that will be generated when the compressed buffer specified by *Source* and *SourceSize* is decompressed.
ScratchSize	A pointer to the size, in bytes, of the scratch buffer that is required to decompress the compressed buffer specified by *Source* and *SourceSize*.

Description

The **GetInfo()** function retrieves the size of the uncompressed buffer and the temporary scratch buffer required to decompress the buffer specified by *Source* and *SourceSize*. If the size of the uncompressed buffer or the size of the scratch buffer cannot be determined from the compressed data specified by *Source* and *SourceData*, then **EFI_INVALID_PARAMETER** is returned. Otherwise, the size of the uncompressed buffer is returned in *DestinationSize*, the size of the scratch buffer is returned in *ScratchSize*, and **EFI_SUCCESS** is returned.

The **GetInfo()** function does not have scratch buffer available to perform a thorough checking of the validity of the source data. It just retrieves the "Original Size" field from the beginning bytes of the source data and output it as *DestinationSize*. And *ScratchSize* is specific to the decompression implementation.

Status Codes Returned

EFI_SUCCESS	The size of the uncompressed data was returned in *DestinationSize* and the size of the scratch buffer was returned in *ScratchSize*.
EFI_INVALID_PARAMETER	The size of the uncompressed data or the size of the scratch buffer cannot be determined from the compressed data specified by *Source* and *SourceData*.

EFI_DECOMPRESS_PROTOCOL.Decompress()

Summary

Decompresses a compressed source buffer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DECOMPRESS_DECOMPRESS) (
  IN    EFI_DECOMPRESS_PROTOCOL   *This,
  IN      VOID*                   Source,
  IN      UINT32                  SourceSize,
  IN OUT VOID*                    Destination,
  IN      UINT32                  DestinationSize,
  IN OUT VOID*                    Scratch,
  IN      UINT32                  ScratchSize
  );
```

Parameters

This	A pointer to the **EFI_DECOMPRESS_PROTOCOL** instance. Type **EFI_DECOMPRESS_PROTOCOL** is defined in Section 17.5.
Source	The source buffer containing the compressed data.
SourceSize	The size of source data.
Destination	On output, the destination buffer that contains the uncompressed data.
DestinationSize	The size of the destination buffer. The size of the destination buffer needed is obtained from **GetInfo()**.
Scratch	A temporary scratch buffer that is used to perform the decompression.
ScratchSize	The size of scratch buffer. The size of the scratch buffer needed is obtained from **GetInfo()**.

Description

The **Decompress()** function extracts decompressed data to its original form.

This protocol is designed so that the decompression algorithm can be implemented without using any memory services. As a result, the **Decompress()** function is not allowed to call **AllocatePool()** or **AllocatePages()** in its implementation. It is the caller's responsibility to allocate and free the *Destination* and *Scratch* buffers.

If the compressed source data specified by *Source* and *SourceSize* is sucessfully decompressed into *Destination*, then **EFI_SUCCESS** is returned. If the compressed source data specified by *Source* and *SourceSize* is not in a valid compressed data format, then **EFI_INVALID_PARAMETER** is returned.

Status Codes Returned

EFI_SUCCESS	Decompression completed successfully, and the uncompressed buffer is returned in *Destination*.
EFI_INVALID_PARAMETER	The source buffer specified by *Source* and *SourceSize* is corrupted (not in a valid compressed format).

18 Protocols - Device I/O Protocol

This chapter defines the Device I/O protocol. This protocol is used by code, typically drivers, running in the EFI boot services environment to access memory and I/O. In particular, functions for managing PCI buses are defined here although other bus types may be supported in a similar fashion as extensions to this specification.

The services defined in this chapter have been superceded by the services described in Chapter 12. Both the PCI Root Bridge I/O Protocol and the PCI I/O Protocol provide a more complete set of services for managing PCI devices. The PCI Root Bridge I/O Protocol and PCI I/O Protocol are not defined in the *EFI 1.02 Specification*. If an EFI image is required to be compliant with the *EFI 1.02 Specification*, then the Device I/O Protocol is the only option for these types of I/O services. If an EFI image is required to be compliant with the *EFI 1.10 Specification*, then the PCI Root Bridge I/O Protocol or the PCI I/O Protocol must be used for these types of I/O services.

18.1 Device I/O Overview

The interfaces provided in the **DEVICE_IO** protocol are for performing basic operations to memory, I/O, and PCI configuration space. The **DEVICE_IO** protocol can be thought of as the bus driver for the system. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources.

The **DEVICE_IO** protocol allows for future innovation of the platform. It abstracts device-specific code from the system memory map. This allows system designers to greatly change the system memory map without impacting platform independent code that is consuming basic system resources.

It is important to note that this specification ties these interfaces into a single protocol solely for the purpose of simplicity. Other similar bus- or device-specific protocols that "programmatic child drivers" may require can easily be added by using a new protocol GUID. For example, a comprehensive USB-specific host controller protocol interface could be defined for child drivers. These drivers would perform a **LocateDevicePath()** to obtain the proper USB interface set, from somewhere up the device path, just as a PCI-based device driver would do with the **DEVICE_IO** protocol to gain access to the PCI configuration space interfaces.

18.2 DEVICE_IO Protocol

This section defines the Device I/O Protocol. This protocol provides the basic Memory, I/O, and PCI interfaces that are used to abstract accesses to devices.

DEVICE_IO Protocol

Summary

Provides the basic Memory, I/O, and PCI interfaces that are used to abstract accesses to devices.

GUID

```
#define DEVICE_IO_PROTOCOL \
     { af6ac311-84c3-11d2-8e3c-00a0c969723b }
```

Protocol Interface Structure

```
typedef struct _EFI_DEVICE_IO_INTERFACE {
     EFI_IO_ACCESS                          Mem;
     EFI_IO_ACCESS                          Io;
     EFI_IO_ACCESS                          Pci;
     EFI_IO_MAP                             Map;
     EFI_PCI_DEVICE_PATH                    PciDevicePath;
     EFI_IO_UNMAP                           Unmap;
     EFI_IO_ALLOCATE_BUFFER                 AllocateBuffer;
     EFI_IO_FLUSH                           Flush;
     EFI_IO_FREE_BUFFER                     FreeBuffer;
} EFI_DEVICE_IO_INTERFACE;
```

Parameters

Mem	Allows reads and writes to memory mapped I/O space. See the **Mem()** function description.
Io	Allows reads and writes to I/O space. See the **Io()** function description.
Pci	Allows reads and writes to PCI configuration space. See the **Pci()** function description.
Map	Provides the device specific addresses needed to access system memory for DMA. See the **Map()** function description.
PciDevicePath	Provides an EFI Device Path for a PCI device with the given PCI configuration space address. See the **PciDevicePath()** function description.

Unmap	Releases any resources allocated by **Map()**. See the **Unmap()** function description.
AllocateBuffer	Allocates pages that are suitable for a common buffer mapping. See the **AllocateBuffer()** function description.
Flush	Flushes any posted write data to the device. See the **Flush()** function description.
FreeBuffer	Free pages that were allocated with **AllocateBuffer()**. See the **FreeBuffer()** function description.

Related Definitions

```
//*******************************************************
// EFI_IO_WIDTH
//*******************************************************

typedef enum {
     IO_UINT8  = 0,
     IO_UINT16 = 1,
     IO_UINT32 = 2,
     IO_UINT64 = 3
} EFI_IO_WIDTH;

//*******************************************************
// EFI_DEVICE_IO
//*******************************************************

typedef
EFI_STATUS
(EFIAPI *EFI_DEVICE_IO) (
  IN struct _EFI_DEVICE_IO_INTERFACE   *This,
  IN EFI_IO_WIDTH                      Width,
  IN UINT64                            Address,
  IN UINTN                             Count,
  IN OUT VOID                          *Buffer
  );

//*******************************************************
// EFI_IO_ACCESS
//*******************************************************

typedef struct {
     EFI_DEVICE_IO                          Read;
     EFI_DEVICE_IO                          Write;
} EFI_IO_ACCESS;
```

Description

The **DEVICE_IO** protocol provides the basic Memory, I/O, and PCI interfaces that are used to abstract accesses to devices.

A driver that controls a physical device obtains the proper **DEVICE_IO** protocol interface by checking for the supported protocol on the programmatic parent(s) for the device. This is easily done via the **LocateDevicePath()** boot service function.

The following C code fragment illustrates the use of the **DEVICE_IO** protocol:

```
// Get the handle to our parent that provides the device I/O
// protocol interfaces to "MyDevice" (which has the device path
// of "MyDevicePath")
EFI_DEVICE_IO_INTERFACE       *IoFncs;
EFI_DEVICE_PATH               *SearchPath;

SearchPath = MyDevicePath;
Status = LocateDevicePath (
                 &DeviceIoProtocol,         // Protocol GUID
                 &SearchPath,               // Device Path SearchKey
                 &DevHandle                 // Return EFI Handle
                 );

// Get the device I/O interfaces from the handle
Status = HandleProtocol (DevHandle, &DeviceIoProtocol, &IoFncs);

// Read 1 dword into Buffer from MyDevice's I/O address
IoFncs->Io.Read (IoFncs, IO_UINT32, MyDeviceAddress, 1, &Buffer);
```

The call to **LocateDevicePath()** takes the Device Path of a device and returns the handle that contains the **DEVICE_IO** protocol for the device. The handle is passed to **HandleProtocol()** with a pointer to the **EFI_GUID** for the **DEVICE_IO** protocol, and a pointer to the **DEVICE_IO** protocol is returned. The **DEVICE_IO** protocol pointer **IoFncs** is then used to do an I/O read to a device.

DEVICE_IO.Mem(), .Io(), and .Pci()

Summary

Enables a driver to access device registers in the appropriate memory or I/O space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DEVICE_IO) (
  IN struct EFI_DEVICE_IO_INTERFACE  *This,
  IN EFI_IO_WIDTH                    Width,
  IN UINT64                          Address,
  IN UINTN                           Count,
  IN OUT VOID                        *Buffer
  );
```

Parameters

This	A pointer to the **EFI_DEVICE_IO_INTERFACE** instance. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 18.2.
Width	Signifies the width of the I/O operations. Type **EFI_IO_WIDTH** is defined in Section 18.2.
Address	The base address of the I/O operations. The caller is responsible for aligning the *Address* if required.
Count	The number of I/O operations to perform. Bytes moved is *Width* size * *Count*, starting at *Address*.
Buffer	For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description

The **DEVICE_IO.Mem()**, **.Io()**, and **.Pci()** functions enable a driver to access device registers in the appropriate memory or I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues which the bus, device, platform, or type of I/O might require. For example on IA-32 platforms, width requests of **IO_UINT64** do not work.

For **Mem()** and **Io()**, the address field is the bus relative address as seen by the device on the bus. For **Mem()** and **Io()** the caller must align the starting address to be on a proper width boundary.

For **Pci()**, the address field is encoded as shown in Table 18-1. The caller must align the register number being accessed to be on a proper width boundary.

Table 18-1. PCI Address

Mnemonic	Byte Offset	Byte Length	Description
Register	0	1	The register number on the function.
Function	1	1	The function on the device.
Device	2	1	The device on the bus.
Bus	3	1	The bus.
Segment	4	1	The segment number.
Reserved	5	3	Must be zero.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the device.
EFI_INVALID_PARAMETER	*Width* is invalid.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

DEVICE_IO.PciDevicePath()

Summary

Provides an EFI Device Path for a PCI device with the given PCI configuration space address.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_DEVICE_PATH) (
  IN EFI_DEVICE_IO_INTERFACE      *This,
  IN UINT64                       PciAddress,
  IN OUT EFI_DEVICE_PATH          **PciDevicePath
  );
```

Parameters

This	A pointer to the **EFI_DEVICE_IO_INTERFACE** instance. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 18.2.
PciAddress	The PCI configuration space address of the device whose Device Path is going to be returned. The address field is encoded as shown in Table 18-1.
PciDevicePath	A pointer to the pointer for the EFI Device Path for *PciAddress*. Memory for the Device Path is allocated from the pool. Type **EFI_DEVICE_PATH** is defined in Chapter 8.

Description

The **DEVICE_IO.PciDevicePath()** function provides an EFI Device Path for a PCI device with the given PCI configuration space address.

A Device Path for the requested PCI device is returned in *PciDevicePath*. **PciDevicePath()** allocates the memory required for the Device Path from the pool and the caller is responsible for calling **FreePool()** to free the memory used to contain the Device Path. If there is not enough memory to calculate or return the *PciDevicePath* the function will return **EFI_OUT_OF_RESOURCES**. If the function cannot calculate a valid Device Path for *PciAddress* the function will return **EFI_UNSUPPORTED**.

Status Codes Returned

EFI_SUCCESS	The *PciDevicePath* returns a pointer to a valid EFI Device Path.
EFI_UNSUPPORTED	The *PciAddress* does not map to a valid EFI Device Path.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

DEVICE_IO.Map()

Summary

Provides the device-specific addresses needed to access system memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IO_MAP) (
  IN EFI_DEVICE_IO_INTERFACE      *This,
  IN EFI_IO_OPERATION_TYPE        Operation,
  IN EFI_PHYSICAL_ADDRESS         *HostAddress,
  IN OUT UINTN                    *NumberOfBytes,
  OUT EFI_PHYSICAL_ADDRESS        *DeviceAddress,
  OUT VOID                        **Mapping
  );
```

Parameters

This	A pointer to the **EFI_DEVICE_IO_INTERFACE** instance. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 18.2.
Operation	Indicates if the bus master is going to read or write to system memory. Type **EFI_IO_OPERATION_TYPE** is defined in "Related Definitions" below.
HostAddress	The system memory address to map to the device. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5.
NumberOfBytes	On input the number of bytes to map. On output the number of bytes that were mapped.
DeviceAddress	The resulting map address for the bus master device to use to access the hosts *HostAddress*. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5.
Mapping	A resulting value to pass to **Unmap()**.

Related Definitions

```
//*******************************************************
// EFI_IO_OPERATION_TYPE
//*******************************************************
     typedef enum {
          EfiBusMasterRead,
          EfiBusMasterWrite,
          EfiBusMasterCommonBuffer
     } EFI_IO_OPERATION_TYPE;
```

`EfiBusMasterRead`	A read operation from system memory by a bus master.
`EfiBusMasterWrite`	A write operation to system memory by a bus master.
`EfiBusMasterCommonBuffer`	Provides both read and write access to system memory by both the processor and a bus master. The buffer is coherent from both the processor's and the bus master's point of view.

Description

The **`DEVICE_IO.Map()`** function provides the device specific addresses needed to access system memory. This function is used to map system memory for bus master DMA accesses.

All bus master accesses must be performed through their mapped addresses and such mappings must be freed with **`Unmap()`** when complete. If the bus master access is a single read or write data transfer, then **`EfiBusMasterRead`** or **`EfiBusMasterWrite`** is used and the range is unmapped to complete the operation. If performing an **`EfiBusMasterRead`** operation, all the data must be present in system memory before the **`Map()`** is performed. Similarly, if performing an **`EfiBusMasterWrite`**, the data cannot be properly accessed in system memory until the **`Unmap()`** is performed.

Bus master operations that require both read and write access or require multiple host device interactions within the same mapped region must use **`EfiBusMasterCommonBuffer`**. However, only memory allocated via the **`DEVICE_IO.AllocateBuffer()`** interface is guaranteed to be able to be mapped for this operation type.

In all mapping requests the resulting *`NumberOfBytes`* actually mapped may be less than requested.

Status Codes Returned

EFI_SUCCESS	The range was mapped for the returned *`NumberOfBytes`*.
EFI_INVALID_PARAMETER	The *`Operation`* or *`HostAddress`* is undefined.
EFI_UNSUPPORTED	The *`HostAddress`* cannot be mapped as a common buffer.
EFI_DEVICE_ERROR	The system hardware could not map the requested address.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

DEVICE_IO.Unmap()

Summary

Completes the **Map()** operation and releases any corresponding resources.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IO_UNMAP) (
  IN EFI_DEVICE_IO_INTERFACE    *This,
  IN VOID                       *Mapping
  );
```

Parameters

This	A pointer to the **EFI_DEVICE_IO_INTERFACE** instance. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 18.2.
Mapping	The mapping value returned from **Map()**.

Description

The **Unmap()** function completes the **Map()** operation and releases any corresponding resources. If the operation was an **EFIBusMasterWrite**, the data is committed to the target system memory. Any resources used for the mapping are freed.

Status Codes Returned

EFI_SUCCESS	The range was unmapped.
EFI_DEVICE_ERROR	The data was not committed to the target system memory.

DEVICE_IO.AllocateBuffer()

Summary

Allocates pages that are suitable for an **EFIBusMasterCommonBuffer** mapping.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IO_ALLOCATE_BUFFER) (
  IN EFI_DEVICE_IO_INTERFACE     *This,
  IN EFI_ALLOCATE_TYPE           Type,
  IN EFI_MEMORY_TYPE             MemoryType,
  IN UINTN                       Pages,
  IN OUT EFI_PHYSICAL_ADDRESS    *HostAddress
  );
```

Parameters

This	A pointer to the **EFI_DEVICE_IO_INTERFACE** instance. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 18.2.
Type	The type allocation to perform. Type **EFI_ALLOCATE_TYPE** is defined in Chapter 5.
MemoryType	The type of memory to allocate, **EfiBootServicesData** or **EfiRuntimeServicesData**. Type **EFI_MEMORY_TYPE** is defined in Chapter 5.
Pages	The number of pages to allocate.
HostAddress	A pointer to store the base address of the allocated range. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5.

Description

The **AllocateBuffer()** function allocates pages that are suitable for an **EFIBusMasterCommonBuffer** mapping.

The **AllocateBuffer()** function internally calls **AllocatePages()** to allocate a memory range that can be mapped as an **EFIBusMasterCommonBuffer**. When the buffer is no longer needed, the driver frees the memory with a call to **FreeBuffer()**.

Allocation requests of *Type* **AllocateAnyPages** will allocate any available range of pages that satisfies the request. On input the data pointed to by *HostAddress* is ignored.

Allocation requests of *Type* **AllocateMaxAddress** will allocate any available range of pages that satisfies the request that are below or equal to the value pointed to by *HostAddress* on input. On success, the value pointed to by *HostAddress* contains the base of the range actually allocated. If there are not enough consecutive available pages below the requested address, an error is returned.

Allocation requests of *Type* **AllocateAddress** will allocate the pages at the address supplied in the data pointed to by *HostAddress*. If the range is not available memory an error is returned.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were allocated.
EFI_OUT_OF_RESOURCES	The memory pages could not be allocated.
EFI_INVALID_PARAMETER	The requested memory type is invalid.
EFI_UNSUPPORTED	The requested *HostAddress* is not supported on this platform.

DEVICE_IO.Flush()

Summary

Flushes any posted write data to the device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IO_FLUSH) (
  IN EFI_DEVICE_IO_INTERFACE     *This
  );
```

Parameters

This	A pointer to the **EFI_DEVICE_IO_INTERFACE** instance. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 18.2.

Description

The **Flush()** function flushes any posted write data to the device.

Status Codes Returned

EFI_SUCCESS	The buffers were flushed.
EFI_DEVICE_ERROR	The buffers were not flushed due to a hardware error.

DEVICE_IO.FreeBuffer()

Summary

Frees pages that were allocated with **AllocateBuffer()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IO_FREE_BUFFER) (
  IN EFI_DEVICE_IO_INTERFACE     *This,
  IN UINTN                       Pages,
  IN EFI_PHYSICAL_ADDRESS        HostAddress
  );
```

Parameters

This	A pointer to the **EFI_DEVICE_IO_INTERFACE** instance. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 18.2.
Pages	The number of pages to free.
HostAddress	The base address of the range to free. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5.

Description

The **FreeBuffer()** function frees pages that were allocated with **AllocateBuffer()**.

The **FreeBuffer()** function internally calls **FreePages()** to free a memory range.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were freed.
EFI_NOT_FOUND	The requested memory pages were not allocated with **AllocateBuffer()**.
EFI_INVALID_PARAMETER	*HostAddress* is not page aligned or *Pages* is invalid.

19 EFI Byte Code Virtual Machine

This chapter defines an EFI Byte Code (EBC) Virtual Machine that can provide platform- and processor-independent mechanisms for loading and executing EFI device drivers.

19.1 Overview

The current design for option ROMs that are used in IA-32 personal computer systems has been in place since 1981. Attempts to change the basic design requirements have failed for a variety of reasons. The EBC Virtual Machine described in this chapter is attempting to help achieve the following goals:

- Abstract and extensible design
- Processor independence
- OS independence
- Build upon existing specifications when possible
- Facilitate the removal of legacy infrastructure
- Exclusive use of EFI Services

One way to satisfy many of these goals is to define a pseudo or virtual machine that can interpret a predefined instruction set. This will allow the virtual machine to be ported across processor and system architectures without changing or recompiling the option ROM. This specification defines a set of machine level instructions that can be generated by a C compiler.

The following sections are a detailed description of the requirements placed on future option ROMs.

19.1.1 Processor Architecture Independence

Option ROM images shall be independent of IA-32 and Itanium architectures. In order to abstract the architectural differences between processors (not just limited to IA-32 and Itanium processors) option ROM images shall be EBC. This model is presented below:

- 64-bit C source code
- The EFI EBC image is the flashed image
- The system BIOS implements the EBC interpreter
- The interpreter handles 32 vs. 64 bit issues

Current Option ROM technology is processor dependent and heavily reliant upon the existence of the PC-AT infrastructure. These dependencies inhibit the evolution of both hardware and software under the veil of “backward compatibility.” A solution that isolates the hardware and support infrastructure through abstraction will facilitate the uninhibited progression of technology.

19.1.2 OS Independent

Option ROMs shall not require or assume the existence of a particular OS.

19.1.3 EFI Compliant

Option ROM compliance with EFI requires (but is not limited to) the following:

1. Little endian layout
2. Single-threaded model with interrupt polling if needed
3. Where EFI provides required services, EFI is used exclusively. These include:
    - Console I/O
    - Memory Management
    - Timer services
    - Global variable access
4. When an Option ROM provides EFI services, the EFI specification is strictly followed:
    - Service/protocol installation
    - Calling conventions
    - Data structure layouts
    - Guaranteed return on services

19.1.4 Coexistence of Legacy Option ROMs

The infrastructure shall support coexistent Legacy Option ROM and EBC Option ROM images. This case would occur, for example, when a Plug and Play Card has both Legacy and EBC Option ROM images flashed. The details of the mechanism used to select which image to load is beyond the scope of this document. Basically, a legacy System BIOS would not recognize an EBC Option ROM and therefore would never load it. Conversely, an EFI Firmware Boot Manager would only load images that it supports.

The EBC Option ROM format must utilize a legacy format to the extent that a Legacy System BIOS can:

1. Determine the type of the image, in order to ignore the image. The type must be incompatible with currently defined types.
2. Determine the size of the image, in order to skip to the next image.

19.1.5 Relocatable Image

An EBC option ROM image shall be eligible for placement in any system memory area large enough to accommodate it.

Current option ROM technology requires images to be shadowed in system memory address range 0xC0000 to 0xEFFFF on a 2048 byte boundary. This dependency not only limits the number of Option ROMs, it results in unused memory fragments up to 2 KB.

19.1.6 Size Restrictions Based on Memory Available

EBC option ROM images shall not be limited to a predetermined fixed maximum size.

Current option ROM technology limits the size of a preinitialization option ROM image to 128 KB (126 KB actual). Additionally, in the DDIM an image is not allowed to grow during initialization. It is inevitable that 64-bit solutions will increase in complexity and size. To avoid revisiting this issue, EBC option ROM size is only limited by available system memory. EFI memory allocation services allow device drivers to claim as much memory as they need, within limits of available system memory.

The PCI specification limits the size of an image stored in an option ROM to 16 MB. If the driver is stored on the hard drive then the 16MB option ROM limit does not apply. In addition, the PE/COFF object format limits the size of images to 2 GB.

19.2 Memory Ordering

The term memory ordering refers to the order in which a processor issues reads (loads) and writes (stores) out onto the bus to system memory. The EBC Virtual Machine enforces strong memory ordering, where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances.

19.3 Virtual Machine Registers

The EBC virtual machine utilizes a simple register set. There are two categories of VM registers: general purpose registers and dedicated registers. All registers are 64-bits wide. There are eight (8) general-purpose registers (**R0**-**R7**), which are used by most EBC instructions to manipulate or fetch data. Table 19-1 lists the general-purpose registers in the VM and the conventions for their usage during execution.

Table 19-1. General Purpose VM Registers

Index	Register	Description
0	**R0**	Points to the top of the stack
1-3	**R1-R3**	Preserved across calls
4-7	**R4-R7**	Scratch, not preserved across calls

Register **R0** is used as a stack pointer and is used by the CALL, RET, PUSH, and POP instructions. The VM initializes this register to point to the incoming arguments when an EBC image is started or entered. This register may be modified like any other general purpose VM register using EBC instructions. Register **R7** is used for function return values.

Unlike the general-purpose registers, the VM dedicated registers have specific purposes. There are two dedicated registers: the instruction pointer (**IP**), and the flags (**Flags**) register. Specialized instructions provide access to the dedicated registers. These instructions reference the particular dedicated register by its assigned index value. Table 19-2 lists the dedicated registers and their corresponding index values.

Table 19-2. Dedicated VM Registers

Index	Register	Description	
0	**FLAGS**		
		Bit	**Description**
		0	C = Condition code
		1	SS = Single step
		2..63	Reserved
1	**IP**	Points to current instruction	
2..7	Reserved	Not defined	

The VM **Flags** register contains VM status and context flags. Table 19-3 lists the descriptions of the bits in the **Flags** register.

Table 19-3. VM Flags Register

Bit	Flag	Description
0	C	Condition code. Set to 1 if the result of the last compare was true, or set to 0 if the last compare was false. Used by conditional JMP instructions.
1	S	Single-step. If set, causes the VM to generate a single-step exception after executing each instruction. The bit is not cleared by the VM following the exception.
2..63	-	Reserved

The VM **IP** register is used as an instruction pointer and holds the address of the currently executing EBC instruction. The virtual machine will update the **IP** to the address of the next instruction on completion of the current instruction, and will continue execution from the address indicated in **IP**. The **IP** register can be moved into any general-purpose register (**R0**-**R7**). Data manipulation and data movement instructions can then be used to manipulate the value. The only instructions that may modify the **IP** are the JMP, CALL, and RET instructions. Since the instruction set is designed to use words as the minimum instruction entity, the low order bit (bit 0) of **IP** is always cleared to 0. If a JMP, CALL, or RET instruction causes bit 0 of **IP** to be set to 1, then an alignment exception occurs.

19.4 Natural Indexing

The natural indexing mechanism is the critical functionality that enables EBC to be executed unchanged on 32- or 64-bit systems. Natural indexing is used to specify the offset of data relative to a base address. However, rather than specifying the offset as a fixed number of bytes, the offset is encoded in a form that specifies the actual offset in two parts: a constant offset, and an offset specified as a number of natural units (where one natural unit = sizeof (VOID *)). These two values are used to compute the actual offset to data at runtime. When the VM decodes an index during execution, the resultant offset is computed based on the natural processor size. The encoded indexes themselves may be 16, 32, or 64 bits in size. Table 19-4 describes the fields in a natural index encoding.

Table 19-4. Index Encoding

Bit #	Description
N	Sign bit (sign), most significant bit
N-3..N-1	Bits assigned to natural units (w)
A..N-4	Constant units (c)
0..A-1	Natural units (n)

As shown in Table 19-4, for a given encoded index, the most significant bit (bit N) specifies the sign of the resultant offset after it has been calculated. The sign bit is followed by three bits (N-3..N-1) that are used to compute the width of the natural units field (n). The value (w) from this field is multiplied by the index size in bytes to determine the actual width (A) of the natural units field (n). Once the width of the natural units field has been determined, then the natural units (n) and constant units (c) can be extracted. The offset is then calculated at runtime according to the following equation:

Offset = (c + n * (sizeof (VOID *))) * sign

The following sections describe each of these fields in more detail.

19.4.1 Sign Bit

The sign bit determines the sign of the index once the offset calculation has been performed. All index computations using “n” and “c” are done with positive numbers, and the sign bit is only used to set the sign of the final offset computed.

19.4.2 Bits Assigned to Natural Units

This 3-bit field that is used to determine the width of the natural units field. The units vary based on the size of the index according to Table 19-5. For example, for a 16-bit index, the value contained in this field would be multiplied by 2 to get the actual width of the natural-units field.

Table 19-5. Index Size in Index Encoding

Index Size	Units
16 bits	2 bits
32 bits	4 bits
64 bits	8 bits

19.4.3 Constant

The constant is the number of bytes in the index that do not scale with processor size. When the index is a 16-bit value, the maximum constant is 4095. This index is achieved when the bits assigned to natural units is 0.

19.4.4 Natural Units

Natural units are used when a structure has fields that can vary with the architecture of the processor. Fields that precipitate the use of natural units include pointers and EFI INTN and UINTN data types. The size of one pointer or INTN/UINTN equals one natural unit. The natural units field in an index encoding is a count of the number of natural fields whose sizes (in bytes) must be added to determine a field offset.

As an example, assume that a given EBC instruction specifies a 16-bit index of 0xA048. This breaks down into:

- Sign bit (bit 15) = 1 (negative offset)
- Bits assigned to natural units (w, bits 14-12) = 2. Multiply by index size in bytes = 2 x 2 = 4 (A)
- c = bits 11-4 = 4
- n = bits 3-0 = 8

On a 32-bit machine, the offset is then calculated to be:

- Offset = (4 + 8 * 4) * -1 = -36

On a 64-bit machine, the offset is calculated to be:

- Offset = (4 + 8 * 8) * -1 = -68

19.5 EBC Instruction Operands

The VM supports an EBC instruction set that performs data movement, data manipulation, branching, and other miscellaneous operations typical of a simple processor. Most instructions operate on two operands, and have the general form:

INSTRUCTION Operand1, Operand2

Typically, instruction operands will be one of the following:

- Direct
- Indirect
- Indirect with index
- Immediate

The following subsections explain these operands.

19.5.1 Direct Operands

When a direct operand is specified for an instruction, the data to operate upon is contained in one of the VM general-purpose registers **R0**-**R7**. Syntactically, an example of direct operand mode could be the ADD instruction:

ADD64 R1, R2

This form of the instruction utilizes two direct operands. For this particular instruction, the VM would take the contents of register **R2**, add it to the contents of register **R1**, and store the result in register **R1**.

19.5.2 Indirect Operands

When an indirect operand is specified, a VM register contains the address of the operand data. This is sometimes referred to as register indirect, and is indicated by prefixing the register operand with "@." Syntactically, an example of an indirect operand mode could be this form of the ADD instruction:

ADD32 R1, @R2

For this instruction, the VM would take the 32-bit value at the address specified in **R2**, add it to the contents of register **R1**, and store the result in register **R1**.

19.5.3 Indirect with Index Operands

When an indirect with index operand is specified, the address of the operand is computed by adding the contents of a register to a decoded natural index that is included in the instruction. Typically with indexed addressing, the base address will be loaded in the register and an index value will be used to indicate the offset relative to this base address. Indexed addressing takes the form

@R_1 (+n,+c)

where:

- **R_1** is one of the general-purpose registers (**R0**-**R7**) which contains the base address
- **+n** is a count of the number of "natural" units offset. This portion of the total offset is computed at runtime as (n * sizeof (VOID *))
- **+c** is a byte offset to add to the natural offset to resolve the total offset

The values of **n** and **c** can be either positive or negative, though they must both have the same sign. These values get encoded in the indexes associated with EBC instructions as shown in Table 19-4. Indexes can be 16-, 32-, or 64-bits wide depending on the instruction. An example of indirect with index syntax would be:

ADD32 R1, @R2 (+1, +8)

This instruction would take the address in register **R2**, add (8 + 1 * sizeof (VOID *)), read the 32-bit value at the address, add the contents of **R1** to the value, and store the result back to **R1**.

19.5.4 Immediate Operands

Some instructions support an immediate operand, which is simply a value included in the instruction encoding. The immediate value may or may not be sign extended, depending on the particular instruction. One instruction that supports an immediate operand is MOVI. An example usage of this instruction is:

MOVIww R1, 0x1234

This instruction moves the immediate value 0x1234 directly into VM register **R1**. The immediate value is contained directly in the encoding for the MOVI instruction.

19.6 EBC Instruction Syntax

Most EBC instructions have one or more variations that modify the size of the instruction and/or the behavior of the instruction itself. These variations will typically modify an instruction in one or more of the following ways:

- The size of the data being operated upon
- The addressing mode for the operands
- The size of index or immediate data

To represent these variations syntactically in this specification the following conventions are used:

- Natural indexes are indicated with the “Index” keyword, and may take the form of “Index16,” “Index32,” or “Index64” to indicate the size of the index value supported. Sometimes the form Index16|32|64 is used here, which is simply a shorthand notation for Index16|Index32|Index64. A natural index is encoded per Table 19-4 and is resolved at runtime.
- Immediate values are indicated with the “Immed” keyword, and may take the form of “Immed16,” “Immed32,” or “Immed64” to indicate the size of the immediate value supported. The shorthand notation Immed16|32|64 is sometimes used when different size immediate values are supported.
- Terms in brackets [ ] are required.
- Terms in braces { } are optional.
- Alternate terms are separated by a vertical bar |.
- The form R_1 and R_2 represent Operand 1 register and Operand 2 register respectfully, and can typically be any VM general-purpose register **R0**-**R7**.
- Within descriptions of the instructions, brackets [ ] enclosing a register and/or index indicate that the contents of the memory pointed to by the enclosed contents are used.

19.7 Instruction Encoding

Most EBC instructions take the form:

INSTRUCTION R_1, R_2 Index|Immed

For those instructions that adhere to this form, the binary encoding for the instruction will typically consist of an opcode byte, followed by an operands byte, followed by two or more bytes of immediate or index data. Thus the instruction stream will be:

(1 Byte Opcode) + (1 Byte Operands) + (Immediate data|Index data)

19.7.1 Instruction Opcode Byte Encoding

The first byte of an instruction is the opcode byte, and an instruction's actual opcode value consumes 6 bits of this byte. The remaining two bits will typically be used to indicate operand sizes and/or presence or absence of index or immediate data. Table 19-6 defines the bits in the opcode byte for most instructions, and their usage.

Table 19-6. Opcode Byte Encoding

Bit	Sym	Description
6..7	Modifiers	One or more of: • Index or immediate data present/absent • Operand size • Index or immediate data size
0..5	Op	Instruction opcode

For those instructions that use bit 7 to indicate the presence of an index or immediate data and bit 6 to indicate the size of the index or immediate data, if bit 7 is 0 (no immediate data), then bit 6 is ignored by the VM. Otherwise, unless otherwise specified for a given instruction, setting unused bits in the opcode byte results in an instruction encoding exception when the instruction is executed. Setting the modifiers field in the opcode byte to reserved values will also result in an instruction encoding exception.

19.7.2 Instruction Operands Byte Encoding

The second byte of most encoded instructions is an operand byte, which encodes the registers for the instruction operands and whether the operands are direct or indirect. Table 19-7 defines the encoding for the operand byte for these instructions. Unless otherwise specified for a given instruction, setting unused bits in the operand byte results in an instruction encoding exception when the instruction is executed. Setting fields in the operand byte to reserved values will also result in an instruction encoding exception.

Table 19-7. Operand Byte Encoding

Bit	Description
7	0 = Operand 2 is direct 1 = Operand 2 is indirect
4..6	Operand 2 register
3	0 = Operand 1 is direct 1 = Operand 1 is indirect
0..2	Operand 1 register

19.7.3 Index/Immediate Data Encoding

Following the operand bytes for most instructions is the instruction's immediate data. The immediate data is, depending on the instruction and instruction encoding, either an unsigned or signed literal value, or an index encoded using natural encoding. In either case, the size of the immediate data is specified in the instruction encoding.

For most instructions, the index/immediate value in the instruction stream is interpreted as a signed immediate value if the register operand is direct. This immediate value is then added to the contents of the register to compute the instruction operand. If the register is indirect, then the data is usually interpreted as a natural index (see Section 19.4) and the computed index value is added to the contents of the register to get the address of the operand.

19.8 EBC Instruction Set

The following sections describe each of the EBC instructions in detail. Information includes an assembly-language syntax, a description of the instruction functionality, binary encoding, and any limitations or unique behaviors of the instruction.

ADD

SYNTAX:

ADD[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Adds two signed operands and stores the result to Operand 1. The operation can be performed on either 32-bit (ADD32) or 64-bit (ADD64) operands.

OPERATION:

Operand 1 <= Operand 1 + Operand 2

Table 19-8. ADD Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0C
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the R_2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is ADD32 and Operand 1 is direct, then the result is stored back to the Operand 1 register with the upper 32 bits cleared.

AND

SYNTAX:

AND[32|64] {@}R$_1$, {@}R$_2$ {Index16|Immed16}

DESCRIPTION:

Performs a logical AND operation on two operands and stores the result to Operand 1. The operation can be performed on either 32-bit (AND32) or 64-bit (AND64) operands.

OPERATION:

Operand 1 <= Operand 1 AND Operand 2

Table 19-9. AND Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x14
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R$_2$ + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the register contents such that Operand 2 = R$_2$ + Immed16.
- If the instruction is AND32 and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

ASHR

SYNTAX:

ASHR[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Performs an arithmetic right-shift of a signed 32-bit (ASHR32) or 64-bit (ASHR64) operand and stores the result back to Operand 1

OPERATION:

Operand 1 <= Operand 1 SHIFT-RIGHT Operand 2

Table 19-10 ASHR Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x19
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2+ Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is ASHR32, and Operand 1 is direct, then the result is stored back to the Operand 1 register with the upper 32 bits cleared.

BREAK

SYNTAX:

BREAK [break code]

DESCRIPTION

The BREAK instruction is used to perform special processing by the VM. The break code specifies the functionality to perform.

BREAK 0 – Runaway program break. This indicates that the VM is likely executing code from cleared memory. This results in a bad break exception.

BREAK 1 – Get virtual machine version. This instruction returns the 64-bit virtual machine revision number in VM register **R7**. The encoding is shown in Table 19-11 and Table 19-12. A VM that conforms to this version of the specification should return a version number of 0x00010000.

Table 19-11 VM Version format

BITS	DESCRIPTION
63-32	Reserved = 0
31..16	VM major version
15..0	VM minor version

BREAK 3 – Debug breakpoint. Executing this instruction results in a debug break exception. If a debugger is attached or available, then it may halt execution of the image.

BREAK 4 – System call. There are no system calls supported for use with this break code, so the VM will ignore the instruction and continue execution at the following instruction.

BREAK 5 – Create thunk. This causes the interpreter to create a thunk for the EBC entry point whose 32-bit IP-relative offset is stored at the 64-bit address in VM register **R7**. The interpreter then replaces the contents of the memory location pointed to by **R7** to point to the newly created thunk. Since all EBC IP-relative offsets are relative to the next instruction or data object, the original offset is off by 4, so must be incremented by 4 to get the actual address of the entry point.

BREAK 6 – Set compiler version. An EBC C compiler can insert this break instruction into an executable to set the compiler version used to build an EBC image. When the VM executes this instruction it takes the compiler version from register **R7** and may perform version compatibility checking. The compiler version number follows the same format as the VM version number returned by the BREAK 1 instruction.

Table 19-12. BREAK Instruction Encoding

BYTE	DESCRIPTION
0	Opcode = 0x00
1	0 = Runaway program break 1 = Get virtual machine version 3 = Debug breakpoint 4 = System call 5 = Create thunk 6 = Set compiler version

BEHAVIORS AND RESTRICTIONS:

- Executing an undefined BREAK code results in a bad break exception.
- Executing BREAK 0 results in a bad break exception.

CALL

SYNTAX:

CALL32{EX}{a} {@}R_1 {Immed32|Index32}

CALL64{EX}{a} Immed64

DESCRIPTION:

The CALL instruction pushes the address of the following instruction on the stack and jumps to a subroutine. The subroutine may be either EBC or native code, and may be to an absolute or **IP-**relative address. CALL32 is used to jump directly to EBC code within a given application, whereas CALLEX is used to jump to external code (either native or EBC), which requires thunking. Functionally, the CALL does the following:

```
R0 = R0 - 8;
PUSH64 ReturnAddress
if (Opcode.ImmedData64Bit) {
  if (Operands.EbcCall) {
    IP = Immed64;
  } else {
    NativeCall (Immed64);
  }
} else {
  if (Operand1 != R0) {
    Addr = Operand1;
  } else {
    Addr = Immed32;
  }
  if (Operands.EbcCall) {
    if (Operands.RelativeAddress) {
      IP += Addr + SizeOfThisInstruction;
    } else {
      IP = Addr
    }
  } else {
    if (Operands.RelativeAddress) {
      NativeCall (IP + Addr)
    } else {
      NativeCall (Addr)
    }
  }
```

OPERATION:

R0 <= **R0** – 16

[**R0**] <= **IP** + SizeOfThisInstruction

IP <= **IP** + SizeOfThisInstruction + Operand 1 (relative CALL)

IP <= Operand 1 (absolute CALL)

Table 19-13 CALL Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index data absent 1 = Immediate/index data present
	6	0 = CALL32 with 32-bit immediate data/index if present 1 = CALL64 with 64-bit immediate data
	0..5	Opcode = 0x03
1	**Bit**	**Description**
	6..7	Reserved = 0
	5	0 = Call to EBC 1 = Call to native code
	4	0 = Absolute address 1 = Relative address
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..5	Optional 32-bit index/immediate for CALL32	
2..9	Required 64-bit immediate data for CALL64	

BEHAVIOR AND RESTRICTIONS:

- For the CALL32 forms, if Operand 1 is indirect, then the immediate data is interpreted as an index, and the Operand 1 value is fetched from memory address [R_1 + Index32].
- For the CALL32 forms, if Operand 1 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 1 register contents such that Operand 1 = R_1 + Immed32.
- For the CALLEX forms, the VM must fix up the stack pointer and execute a call to native code in a manner compatible with the native code such that the callee is able to access arguments passed on the VM stack..
- For the CALLEX forms, the value returned by the callee should be returned in **R7**.
- For the CALL64 forms, the Operand 1 fields are ignored.
- If Byte7:Bit6 = 1 (CALL64), then Byte1:Bit4 is assumed to be 0 (absolute address)
- For CALL32 forms, if Operand 1 register = **R0**, then the register operand is ignored and only the immediate data is used in the calculation of the call address.
- Prior to the call, the VM will decrement the stack pointer **R0** by 16 bytes, and store the 64-bit return address on the stack.
- Offsets for relative calls are relative to the address of the instruction following the CALL instruction.

CMP

SYNTAX:

CMP[32|64][eq|lte|gte|ulte|ugte] R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

The CMP instruction is used to compare Operand 1 to Operand 2. Supported comparison modes are =, <=, >=, unsigned <=, and unsigned >=. The comparison size can be 32 bits (CMP32) or 64 bits (CMP64). The effect of this instruction is to set or clear the condition code bit in the **Flags** register per the comparison results. The operands are compared as signed values except for the CMPulte and CMPugte forms.

OPERATION:

CMPeq: **Flags.C** <= (Operand 1 == Operand 2)

CMPlte: **Flags.C** <= (Operand 1 <= Operand 2)

CMPgte: **Flags.C** <= (Operand 1 >= Operand 2)

CMPulte: **Flags.C** <= (Operand 1 <= Operand 2) (unsigned)

CMPugte: **Flags.C** <= (Operand 1>= Operand 2) (unsigned)

Table 19-14 CMP Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index data absent 1 = Immediate/index data present
	6	0 = 32-bit comparison 1 = 64-bit comparison
	0..5	**Opcode**
		0x05 = CMPeq compare equal 0x06 = CMPlte compare signed less then/equal 0x07 = CMPgte compare signed greater than/equal 0x08 = CMPulte compare unsigned less than/equal 0x09 = CMPugte compare unsigned greater than/equal
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	Reserved = 0
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the register contents such that Operand 2 = R_2 + Immed16.
- Only register direct is supported for Operand 1.

CMPI

SYNTAX:

CMPI[32|64]{w|d}[eq|lte|gte|ulte|ugte] {@}R_1 {Index16}, Immed16|Immed32

DESCRIPTION:

Compares two operands, one of which is an immediate value, for =, <=, >=, unsigned <=, or unsigned >=, and sets or clears the condition flag bit in the **Flags** register accordingly. Comparisons can be performed on a 32-bit (CMPI32) or 64-bit (CMPI64) basis. The size of the immediate data can be either 16 bits (CMPIw) or 32 bits (CMPId).

OPERATION:

CMPIeq: **Flags.C** <= (Operand 1 == Operand 2)

CMPIlte: **Flags.C** <= (Operand 1 <= Operand 2)

CMPIgte: **Flags.C** <= (Operand 1 >= Operand 2)

CMPIulte: **Flags.C** <= (Operand 1 <= Operand 2)

CMPIugte: **Flags.C** <= (Operand 1>= Operand 2)

Table 19-15. CMPI Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = 16-bit immediate data 1 = 32-bit immediate data
	6	0 = 32-bit comparison 1 = 64-bit comparison
	0..5	**Opcode**
		0x2D = CMPIeq compare equal 0x2E = CMPIlte compare signed less then/equal 0x2F = CMPIgte compare signed greater than/equal 0x30 = CMPIulte compare unsigned less than/equal 0x31 = CMPIugte compare unsigned greater than/equal
1	**Bit**	**Description**
	5..7	Reserved = 0
	4	0 = Operand 1 index absent 1 = Operand 1 index present
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 1 index	
2..3/4..5	16-bit immediate data	
2..5/4..7	32-bit immediate data	

BEHAVIORS AND RESTRICTIONS:

- The immediate data is fetched as a signed value.
- If the immediate data is smaller than the comparison size, then the immediate data is sign-extended appropriately.
- If Operand 1 is direct, and an Operand 1 index is specified, then an instruction encoding exception is generated.

DIV

SYNTAX:

DIV[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Performs a divide operation on two signed operands and stores the result to Operand 1. The operation can be performed on either 32-bit (DIV32) or 64-bit (DIV64) operands.

OPERATION:

Operand 1 <= Operand 1 / Operand 2

Table 19-16 DIV Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x10
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2+ Index16].
- If Operand 2 is direct, then the immediate data is considered a signed value and is added to the register contents such that Operand 2 = R_2 + Immed16
- If the instruction is DIV32 form, and Operand 1 is direct, then the upper 32 bits of the result are set to 0 before storing to the Operand 1 register.
- A divide-by-0 exception occurs if Operand 2 = 0.

DIVU

SYNTAX:

DIVU[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Performs a divide operation on two unsigned operands and stores the result to Operand 1. The operation can be performed on either 32-bit (DIVU32) or 64-bit (DIVU64) operands.

OPERATION:

Operand 1 <= Operand 1 / Operand 2

Table 19-17 DIVU Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x11
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the value is fetched from memory as an unsigned value at address [R_2+ Index16].
- If Operand 2 is direct, then the immediate data is considered an unsigned value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16
- For the DIVU32 form, if Operand 1 is direct then the upper 32 bits of the result are set to 0 before storing back to the Operand 1 register.
- A divide-by-0 exception occurs if Operand 2 = 0.

EXTNDB

SYNTAX:

EXTNDB[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Sign-extend a byte value and store the result to Operand 1. The byte can be signed extended to 32 bits (EXTNDB32) or 64 bits (EXTNDB64).

OPERATION:

Operand 1 <= (sign extended) Operand 2

Table 19-18 EXTNDB Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x1A
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the byte Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value, is added to the signed-extended byte from the Operand 2 register, and the byte result is sign extended to 32 or 64 bits.
- If the instruction is EXTNDB32 and Operand 1 is direct, then the 32-bit result is stored in the Operand 1 register with the upper 32 bits cleared.

EXTNDD

SYNTAX:

EXTNDD[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Sign-extend a 32-bit Operand 2 value and store the result to Operand 1. The Operand 2 value can be extended to 32 bits (EXTNDD32) or 64 bits (EXTNDD64).

OPERATION:

Operand 1 <= (sign extended) Operand 2

Table 19-19. EXTNDD Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x1C
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the 32-bit value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value such that Operand 2 = R_2 + Immed16, and the value is sign extended to 32 or 64 bits accordingly.
- If the instruction is EXTNDD32 and Operand 1 is direct, then the result is stored in the Operand 1 register with the upper 32 bits cleared.

EXTNDW

SYNTAX:

EXTNDW[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Sign-extend a 16-bit Operand 2 value and store the result back to Operand 1. The value can be signed extended to 32 bits (EXTNDW32) or 64 bits (EXTNDW64).

OPERATION:

Operand 1 <= (sign extended) Operand 2

Table 19-20. EXTNDW Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x1B
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the word value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value such that Operand 2 = R_2 + Immed16, and the value is sign extended to 32 or 64 bits accordingly.
- If the instruction is EXTNDW32 and Operand 1 is direct, then the 32-bit result is stored in the Operand 1 register with the upper 32 bits cleared.

JMP

SYNTAX:

JMP32{cs|cc} {@}R_1 {Immed32|Index32}

JMP64{cs|cc} Immed64

DESCRIPTION:

The JMP instruction is used to conditionally or unconditionally jump to a relative or absolute address and continue executing EBC instructions. The condition test is done using the condition bit in the VM **Flags** register. The JMP64 form only supports an immediate value that can be used for either a relative or absolute jump. The JMP32 form adds support for indirect addressing of the JMP offset or address. The JMP is implemented as:

```
if (ConditionMet) {
  if (Operand.RelativeJump) {
   IP += Operand1 + SizeOfThisInstruction;
  } else {
    IP = Operand1;
  }
}
```

OPERATION:

IP <= Operand 1 (absolute address)

IP <= **IP** + SizeOfThisInstruction + Operand 1 (relative address)

Table 19-21 JMP Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index data absent 1 = Immediate/index data present
	6	0 = JMP32 1 = JMP64
	0..5	Opcode = 0x01
1	**Bit**	**Description**
	7	0 = Unconditional jump 1 = Conditional jump
	6	0 = Jump if **Flags.C** is clear (cc) 1 = Jump if **Flags.C** is set (cs)
	5	Reserved = 0
	4	0 = Absolute address 1 = Relative address
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..5	Optional 32-bit immediate data/index for JMP32	
2..9	64-bit immediate data for JMP64	

BEHAVIORS AND RESTRICTIONS:

- Operand 1 fields are ignored for the JMP64 forms
- If the instruction is JMP32, and Operand 1 register = **R0**, then the register contents are assumed to be 0.
- If the instruction is JMP32, and Operand 1 is indirect, then the immediate data is interpreted as an index, and the jump offset or address is fetched as a 32-bit signed value from address [R_1 + Index32]
- If the instruction is JMP32, and Operand 1 is direct, then the immediate data is considered a signed immediate value such that Operand 1 = R_1 + Immed32
- If the jump is unconditional, then Byte1:Bit6 (condition) is ignored
- If the instruction is JMP64, and Byte0:Bit7 is clear (no immediate data), then an instruction encoding exception is generated.
- If the instruction is JMP32, and Operand 2 is indirect, then the Operand 2 value is read as a natural value from memory address [R_1 + Index32]
- An alignment check exception is generated if the jump is taken and the target address is odd.

JMP8

SYNTAX:

JMP8{cs|cc} Immed8

DESCRIPTION:

Conditionally or unconditionally jump to a relative offset and continue execution. The offset is a signed one-byte offset specified in the number of words. The offset is relative to the start of the following instruction.

OPERATION:

IP = **IP** + SizeOfThisInstruction + (Immed8 * 2)

Table 19-22 JMP8 Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Unconditional jump 1 = Conditional jump
	6	0 = Jump if **Flags.C** is clear (cc) 1 = Jump if **Flags.C** is set (cs)
	0..5	Opcode = 0x02
1	Immediate data (signed word offset)	

BEHAVIORS AND RESTRICTIONS:

- If the jump is unconditional, then Byte0:Bit6 (condition) is ignored

LOADSP

SYNTAX:

LOADSP [**Flags**], R_2

DESCRIPTION:

This instruction loads a VM dedicated register with the contents of a VM general-purpose register **R0**-**R7**. The dedicated register is specified by its index as shown in Table 19-2.

OPERATION:

Operand 1 <= R_2

Table 19-23. LOADSP Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	6..7	Reserved = 0
	0..5	Opcode = 0x29
1	7	Reserved
	4..6	Operand 2 general purpose register
	3	Reserved
	0..2	Operand 1 dedicated register index

BEHAVIORS AND RESTRICTIONS:

- Attempting to load any register (Operand 1) other than the **Flags** register results in an instruction encoding exception.
- Specifying a reserved dedicated register index results in an instruction encoding exception.
- If Operand 1 is the **Flags** register, then reserved bits in the **Flags** register are not modified by this instruction.

MOD

SYNTAX:

MOD[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Perform a modulus on two signed 32-bit (MOD32) or 64-bit (MOD64) operands and store the result to Operand 1.

OPERATION:

Operand 1 <= Operand 1 MOD Operand 2

Table 19-24. MOD Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x12
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value such that Operand 2 = R_2 + Immed16, and the value is sign extended to 32 or 64 bits accordingly.
- If Operand 2 = 0, then a divide-by-zero exception is generated.

MODU

SYNTAX:

MODU[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Perform a modulus on two unsigned 32-bit (MODU32) or 64-bit (MODU64) operands and store the result to Operand 1.

OPERATION:

Operand 1 <= Operand 1 MOD Operand 2

Table 19-25. MODU Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x13
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered an unsigned immediate value such that Operand 2 = R_2 + Immed16.
- If Operand 2 = 0, then a divide-by-zero exception is generated.

MOV

SYNTAX:

MOV[b|w|d|q]{w|d} {@}R_1 {Index16|32}, {@}R_2 {Index16|32}

MOVqq {@}R_1 {Index64}, {@}R_2 {Index64}

DESCRIPTION:

This instruction moves data from Operand 2 to Operand 1. Both operands can be indexed, though both indexes are the same size. In the instruction syntax for the first form, the first variable character indicates the size of the data move, which can be 8 bits (b), 16 bits (w), 32 bits (d), or 64 bits (q). The optional character indicates the presence and size of the index value(s), which may be 16 bits (w) or 32 bits (d). The MOVqq instruction adds support for 64-bit indexes.

OPERATION:

Operand 1 <= Operand 2

Table 19-26. MOV Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 1 index absent 1 = Operand 1 index present
	6	0 = Operand 2 index absent 1 = Operand 2 index present
	0..5	0x1D = MOVbw opcode 0x1E = MOVww opcode 0x1F = MOVdw opcode 0x20 = MOVqw opcode 0x21 = MOVbd opcode 0x22 = MOVwd opcode 0x23 = MOVdd opcode 0x24 = MOVqd opcode 0x28 = MOVqq opcode
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional Operand 1 16-bit index	
2..3/4..5	Optional Operand 2 16-bit index	
2..5	Optional Operand 1 32-bit index	
2..5/6..9	Optional Operand 2 32-bit index	
2..9	Optional Operand 1 64-bit index (MOVqq)	
2..9/10..17	Optional Operand 2 64-bit index (MOVqq)	

BEHAVIORS AND RESTRICTIONS:

- If an index is specified for Operand 1, and Operand 1 is direct, then an instruction encoding exception is generated.

MOVI

SYNTAX:

MOVI[b|w|d|q][w|d|q] {@}R_1 {Index16}, Immed16|32|64

DESCRIPTION:

This instruction moves a signed immediate value to Operand 1. In the instruction syntax, the first variable character specifies the width of the move, which may be 8 bits (b), 16 bits (w), 32-bits (d), or 64 bits (q). The second variable character specifies the width of the immediate data, which may be 16 bits (w), 32 bits (d), or 64 bits (q).

OPERATION:

Operand 1 <= Operand 2

Table 19-27. MOVI Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	6..7	0 = Reserved 1 = Immediate data is 16 bits (w) 2 = Immediate data is 32 bits (d) 3 = Immediate data is 64 bits (q)
	0..5	Opcode = 0x37
1	**Bit**	**Description**
	7	Reserved = 0
	6	0 = Operand 1 index absent 1 = Operand 1 index present
	4..5	0 = 8 bit (b) move 1 = 16 bit (w) move 2 = 32 bit (d) move 3 = 64 bit (q) move
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit index	
2..3/4..5	16-bit immediate data	
2..5/4..7	32-bit immediate data	
2..9/4..11	64-bit immediate data	

BEHAVIORS AND RESTRICTIONS:

- Specifying an index value with Operand 1 direct results in an instruction encoding exception.
- If the immediate data is smaller than the move size, then the value is sign-extended to the width of the move.
- If Operand 1 is a register, then the value is stored to the register with bits beyond the move size cleared.

MOVIn

SYNTAX:

MOVIn[w|d|q] {@}R_1 {Index16}, Index16|32|64

DESCRIPTION:

This instruction moves an indexed value of form (+n,+c) to Operand 1. The index value is converted from (+n, +c) format to a signed offset per the encoding described in Table 19-4. The size of the Operand 2 index data can be 16 (w), 32 (d), or 64 (q) bits.

OPERATION:

Operand 1 <= Operand 2 (index value)

Table 19-28. MOVIn Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	6..7	0 = Reserved 1 = Operand 2 index value is 16 bits (w) 2 = Operand 2 index value is 32 bits (d) 3 = Operand 2 index value is 64 bits (q)
	0..5	Opcode = 0x38
1	**Bit**	**Description**
	7	Reserved
	6	0 = Operand 1 index absent 1 = Operand 1 index present
	4..5	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 1 index	
2..3/4..5	16-bit Operand 2 index	
2..5/4..7	32-bit Operand 2 index	
2..9/4..11	64-bit Operand 2 index	

BEHAVIORS AND RESTRICTIONS:

- Specifying an Operand 1 index when Operand 1 is direct results in an instruction encoding exception.
- The Operand 2 index is sign extended to the size of the move if necessary.
- If the Operand 2 index size is smaller than the move size, then the value is truncated.
- If Operand 1 is direct, then the Operand 2 value is sign extended to 64 bits and stored to the Operand 1 register.

MOVn

SYNTAX:

MOVn{w|d} {@}R_1 {Index16|32}, {@}R_2 {Index16|32}

DESCRIPTION:

This instruction loads an unsigned natural value from Operand 2 and stores the value to Operand 1. Both operands can be indexed, though both operand indexes are the same size. The operand index(s) can be 16 bits (w) or 32 bits (d).

OPERATION:

Operand1 <= (UINTN)Operand2

Table 19-29. MOVn Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 1 index absent 1 = Operand 1 index present
	6	0 = Operand 2 index absent 1 = Operand 2 index present
	0..5	0x32 = MOVnw opcode 0x33 = MOVnd opcode
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional Operand 1 16-bit index	
2..3/4..5	Optional Operand 2 16-bit index	
2..5	Optional Operand 1 32-bit index	
2..5/6..9	Optional Operand 2 32-bit index	

BEHAVIORS AND RESTRICTIONS:

- If an index is specified for Operand 2, and Operand 2 register is direct, then the Operand 2 index value is added to the register contents such that Operand 2 = (UINTN)(R_2 + Index).
- If an index is specified for Operand 1, and Operand 1 is direct, then an instruction encoding exception is generated.
- If Operand 1 is direct, then the Operand 2 value will be 0-extended to 64 bits on a 32-bit machine before storing to the Operand 1 register.

MOVREL

SYNTAX:

MOVREL[w|d|q] {@}R_1 {Index16}, Immed16|32|64

DESCRIPTION:

This instruction fetches data at an **IP**-relative immediate offset (Operand 2) and stores the result to Operand 1. The offset is a signed offset relative to the following instruction. The fetched data is unsigned and may be 16 (w), 32 (d), or 64 (q) bits in size.

OPERATION:

Operand 1 <= [**IP** + SizeOfThisInstruction + Immed]

Table 19-30. MOVREL Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	6..7	0 = Reserved 1 = Immediate data is 16 bits (w) 2 = Immediate data is 32 bits (d) 3 = Immediate data is 64 bits (q)
	0..5	Opcode = 0x39
1	**Bit**	**Description**
	7	Reserved = 0
	6	0 = Operand 1 index absent 1 = Operand 1 index present
	4..5	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 1 index	
2..3/4..5	16-bit immediate offset	
2..5/4..7	32-bit immediate offset	
2..9/4..11	64-bit immediate offset	

BEHAVIORS AND RESTRICTIONS:

- If an Operand 1 index is specified and Operand 1 is direct, then an instruction encoding exception is generated.

MOVsn

SYNTAX:

MOVsn{w} {@}R_1, {Index16}, {@}R_2 {Index16|Immed16}

MOVsn{d} {@}R_1 {Index32}, {@}R_2 {Index32|Immed32}

DESCRIPTION:

Moves a signed natural value from Operand 2 to Operand 1. Both operands can be indexed, though the indexes are the same size. Indexes can be either 16 bits (MOVsnw) or 32 bits (MOVsnd) in size.

OPERATION:

Operand 1 <= Operand 2

Table 19-31. MOVsn Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 1 index absent 1 = Operand 1 index present
	6	0 = Operand 2 index/immediate data absent 1 = Operand 2 index/immediate data present
	0..5	0x25 = MOVsnw opcode 0x26 = MOVsnd opcode
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 1 index (MOVsnw)	
2..3/4..5	Optional 16-bit Operand 2 index (MOVsnw)	
2..5	Optional 32-bit Operand 1 index/immediate data (MOVsnd)	
2..5/6..9	Optional 32-bit Operand 2 index/immediate data (MOVsnd)	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is direct, and Operand 2 index/immediate data is specified, then the immediate value is read as a signed immediate value and is added to the contents of Operand 2 register such that Operand 2 = R_2 + Immed.
- If Operand 2 is indirect, and Operand 2 index/immediate data is specified, then the immediate data is interpreted as an index and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If an index is specified for Operand 1, and Operand 1 is direct, then an instruction encoding exception is generated.
- If Operand 1 is direct, then the Operand 2 value is sign-extended to 64-bits on 32-bit native machines.

MUL

SYNTAX:

MUL[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Perform a signed multiply of two operands and store the result back to Operand 1. The operands can be either 32 bits (MUL32) or 64 bits (MUL64).

OPERATION:

Operand 1 <= Operand * Operand 2

Table 19-32. MUL Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0E
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit Operand 2 immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is MUL32, and Operand 1 is direct, then the result is stored to Operand 1 register with the upper 32 bits cleared.

MULU

SYNTAX:

MULU[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Performs an unsigned multiply of two 32-bit (MULU32) or 64-bit (MULU64) operands, and stores the result back to Operand 1.

OPERATION:

Operand 1 <= Operand * Operand 2

Table 19-33. MULU Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0F
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is MULU32 and Operand 1 is direct, then the result is written to the Operand 1 register with the upper 32 bits cleared.

NEG

SYNTAX:

NEG[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Multiply Operand 2 by negative 1, and store the result back to Operand 1. Operand 2 is a signed value and fetched as either a 32-bit (NEG32) or 64-bit (NEG64) value.

OPERATION

Operand 1 <= -1 * Operand 2

Table 19-34. NEG Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0B
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is NEG32 and Operand 1 is direct, then the result is stored in Operand 1 register with the upper 32-bits cleared.

NOT

SYNTAX:

NOT[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Performs a logical NOT operation on Operand 2, an unsigned 32-bit (NOT32) or 64-bit (NOT64) value, and stores the result back to Operand 1.

OPERATION

Operand 1 <= NOT Operand 2

Table 19-35. NOT Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0A
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is NOT32 and Operand 1 is a register, then the result is stored in the Operand 1 register with the upper 32 bits cleared.

OR

SYNTAX:

OR[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Performs a bit-wise OR of two 32-bit (OR32) or 64-bit (OR64) operands, and stores the result back to Operand 1.

OPERATION:

Operand 1 <= Operand 1 OR Operand 2

Table 19-36. OR Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x15
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is OR32 and Operand 1 is direct, then the result is stored to Operand 1 register with the upper 32 bits cleared.

POP

SYNTAX:

POP[32|64] {@}R_1 {Index16|Immed16}

DESCRIPTION:

This instruction pops a 32-bit (POP32) or 64-bit (POP64) value from the stack, stores the result to Operand 1, and adjusts the stack pointer **R0** accordingly.

OPERATION:

Operand 1 <= [**R0**]

R0 <= **R0** + 4 (POP32)

R0 <= **R0** + 8 (POP64)

Table 19-37. POP Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x2C
1	**Bit**	**Description**
	7..4	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is read as a signed value and is added to the value popped from the stack, and the result stored to the Operand 1 register.
- If Operand 1 is indirect, then the immediate data is interpreted as an index, and the value popped from the stack is stored to address [R_1 + Index16].
- If the instruction is POP32, and Operand 1 is direct, then the popped value is sign-extended to 64 bits before storing to the Operand 1 register.

POPn

SYNTAX:

POPn {@}R_1 {Index16|Immed16}

DESCRIPTION:

Read an unsigned natural value from memory pointed to by stack pointer **R0**, adjust the stack pointer accordingly, and store the value back to Operand 1.

OPERATION:

Operand 1 <= (UINTN)[**R0**]

R0 <= **R0** + sizeof (VOID *)

Table 19-38. POPn Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	Reserved = 0
	0..5	Opcode = 0x36
1	**Bit**	**Description**
	7..4	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is fetched as a signed value and is added to the value popped from the stack and the result is stored back to the Operand 1 register.
- If Operand 1 is indirect, and an index/immediate data is specified, then the immediate data is interpreted as a natural index and the value popped from the stack is stored at [R_1 + Index16].
- If Operand 1 is direct, and the instruction is executed on a 32-bit machine, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

PUSH

SYNTAX:

PUSH[32|64] {@}R_1 {Index16|Immed16}

DESCRIPTION:

Adjust the stack pointer **R0** and store a 32-bit (PUSH32) or 64-bit (PUSH64) Operand 1 value on the stack.

OPERATION:

R0 <= **R0** - 4 (PUSH32)

R0 <= **R0** - 8 (PUSH64)

[**R0**] <= Operand 1

Table 19-39. PUSH Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x2B
1	**Bit**	**Description**
	7..4	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is read as a signed value and is added to the Operand 1 register contents such that Operand 1 = R_1 + Immed16.
- If Operand 1 is indirect, and an index/immediate data is specified, then the immediate data is interpreted as a natural index and the pushed value is read from [R_1 + Index16].

PUSHn

SYNTAX:

PUSHn {@}R_1 {Index16|Immed16}

DESCRIPTION:

Adjust the stack pointer **R0**, and store a natural value on the stack.

OPERATION:

R0 <= **R0** - sizeof (VOID *)

[**R0**] <= Operand 1

Table 19-40. PUSHn Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Immediate/index absent 1 = Immediate/index present
	6	Reserved = 0
	0..5	Opcode = 0x35
1	**Bit**	**Description**
	7..4	Reserved = 0
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is fetched as a signed value and is added to the Operand 1 register contents such that Operand 1 = R_1 + Immed16.
- If Operand 1 is indirect, and an index/immediate data is specified, then the immediate data is interpreted as a natural index and the Operand 1 value pushed is fetched from [R_1 + Index16].

RET

SYNTAX:

RET

DESCRIPTION:

This instruction fetches the return address from the stack, sets the **IP** to the value, adjusts the stack pointer register **R0**, and continues execution at the return address. If the RET is a final return from the EBC driver, then execution control returns to the caller, which may be EBC or native code.

OPERATION:

IP <= [**R0**]

R0 <= **R0** + 16

Table 19-41. RET Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	6..7	Reserved = 0
	0..5	Opcode = 0x04
1	Reserved = 0	

BEHAVIORS AND RESTRICTIONS:

- An alignment exception will be generated if the return address is not aligned on a 16-bit boundary.

SHL

SYNTAX:

SHL[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Left-shifts Operand 1 by Operand 2 bit positions and stores the result back to Operand 1. The operand sizes may be either 32-bits (SHL32) or 64 bits (SHL64).

OPERATION:

Operand 1 <= Operand 1 << Operand 2

Table 19-42. SHL Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x17
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is SHL32, and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

SHR

SYNTAX:

SHR[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Right-shifts unsigned Operand 1 by Operand 2 bit positions and stores the result back to Operand 1. The operand sizes may be either 32-bits (SHR32) or 64 bits (SHR64).

OPERATION:

Operand 1 <= Operand 1 >> Operand 2

Table 19-43. SHR Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x18
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is SHR32, and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

STORESP

SYNTAX:

STORESP R_1, [**IP**|**Flags**]

DESCRIPTION:

This instruction transfers the contents of a dedicated register to a general-purpose register. See Table 19-2 for the VM dedicated registers and their corresponding index values.

OPERATION:

Operand 1 <= Operand 2

Table 19-44. STORESP Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	6..7	Reserved = 0
	0..5	Opcode = 0x2A
1	7	Reserved = 0
	4..6	Operand 2 dedicated register index
	3	Reserved = 0
	0..2	Operand 1 general purpose register

BEHAVIORS AND RESTRICTIONS:

- Specifying an invalid dedicated register index results in an instruction encoding exception.

SUB

SYNTAX:

SUB[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Subtracts a 32-bit (SUB32) or 64-bit (SUB64) signed Operand 2 value from a signed Operand 1 value of the same size, and stores the result to Operand 1.

OPERATION:

Operand 1 <= Operand 1 - Operand 2

Table 19-45. SUB Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x0D
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16
- If the instruction is SUB32 and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

XOR

SYNTAX:

XOR[32|64] {@}R_1, {@}R_2 {Index16|Immed16}

DESCRIPTION:

Performs a bit-wise exclusive OR of two 32-bit (XOR32) or 64-bit (XOR64) operands, and stores the result back to Operand 1.

OPERATION:

Operand 1 <= Operand 1 XOR Operand 2

Table 19-46. XOR Instruction Encoding

BYTE	DESCRIPTION	
0	**Bit**	**Description**
	7	0 = Operand 2 immediate/index absent 1 = Operand 2 immediate/index present
	6	0 = 32-bit operation 1 = 64-bit operation
	0..5	Opcode = 0x16
1	**Bit**	**Description**
	7	0 = Operand 2 direct 1 = Operand 2 indirect
	4..6	Operand 2
	3	0 = Operand 1 direct 1 = Operand 1 indirect
	0..2	Operand 1
2..3	Optional 16-bit immediate data/index	

BEHAVIORS AND RESTRICTIONS:

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R_2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is XOR32 and Operand1 is direct, then the result is stored to the Operand 1 register with the upper 32-bits cleared.

19.9 Runtime and Software Conventions

19.9.1 Calling Outside VM

Calls can be made to routines in other modules that are native or in another VM. It is the responsibility of the calling VM to prepare the outgoing arguments correctly to make the call outside the VM. It is also the responsibility of the VM to prepare the incoming arguments correctly for the call from outside the VM. Calls outside the VM must use the CALLEX instruction.

19.9.2 Calling Inside VM

Calls inside VM can be made either directly using the CALL or CALLEX instructions. Using direct CALL instructions is an optimization.

19.9.3 Parameter Passing

Parameters are pushed on the VM stack per the CDECL calling convention. Per this convention, the last argument in the parameter list is pushed on the stack first, and the first argument in the parameter list is pushed on the stack last.

All parameters are stored or accessed as natural size (using naturally sized instruction) except 64-bit integers, which are pushed as 64-bit values. 32-bit integers are pushed as natural size (since they should be passed as 64-bit parameter values on 64-bit machines).

19.9.4 Return Values

Return values of 8 bytes or less in size are returned in general-purpose register **R7**. Return values larger than 8 bytes are not supported.

19.9.5 Binary Format

PE32+ format will be used for generating binaries for the VM. A VarBss section will be included in the binary image. All global and static variables will be placed in this section. The size of the section will be based on worst-case 64-bit pointers. Initialized data and pointers will also be placed in the VarBss section, with the compiler generating code to initialize the values at runtime.

19.10 Architectural Requirements

This section provides a high level overview of the architectural requirements that are necessary to support execution of EBC on a platform.

19.10.1 EBC Image Requirements

All EBC images will be PE32+ format. Some minor additions to the format will be required to support EBC images. See the *Microsoft Portable Executable and Common Object File Format Specification* pointed to in the References section for details of this image file format.

A given EBC image must be executable on different platforms, independent of whether it is a 32- or 64-bit processor. All EBC images should be driver implementations.

19.10.2 EBC Execution Interfacing Requirements

EBC drivers will typically be designed to execute in an (usually preboot) EFI environment. As such, EBC drivers must be able to invoke protocols and expose protocols for use by other drivers or applications. The following execution transitions must be supported:

- EBC calling EBC
- EBC calling native code
- Native code calling EBC
- Native code calling native code
- Returning from all the above transitions

Obviously native code calling native code is available by default, so is not discussed in this document.

To maintain backward compatibility with existing native code, and minimize the overhead for non-EBC drivers calling EBC protocols, all four transitions must be seamless from the application perspective. Therefore, drivers, whether EBC or native, shall not be required to have any knowledge of whether or not the calling code, or the code being called, is native or EBC compiled code. The onus is put on the tools and interpreter to support this requirement.

19.10.3 Interfacing Function Parameters Requirements

To allow code execution across protocol boundaries, the interpreter must ensure that parameters passed across execution transitions are handled in the same manner as the standard parameter passing convention for the native processor.

19.10.4 Function Return Requirements

The interpreter must support standard function returns to resume execution to the caller of external protocols. The details of this requirement are specific to the native processor. The called function must not be required to have any knowledge of whether or not the caller is EBC or native code.

19.10.5 Function Return Values Requirements

The interpreter must support standard function return values from called protocols. The exact implementation of this functionality is dependent on the native processor. This requirement applies to return values of 64 bits or less. The called function must not be required to have any knowledge of whether or not the caller is EBC or native code. Note that returning of structures is not supported.

19.11 EBC Interpreter Protocol

The EFI EBC protocol provides services to execute EBC images, which will typically be loaded into option ROMs.

EFI_EBC_PROTOCOL

Summary

This protocol provides the services that allow execution of EBC images.

GUID

```
#define EFI_EBC_PROTOCOL_GUID \
  {0x13AC6DD1,0x73D0,0x11D4,0xB0,0x6B,0x00,0xAA,0x00,0xBD,0x6D,0xE7}
```

Protocol Interface Structure

```
typedef struct _EFI_EBC_PROTOCOL {
  EFI_EBC_CREATE_THUNK                      CreateThunk;
  EFI_EBC_UNLOAD_IMAGE                      UnloadImage;
  EFI_EBC_REGISTER_ICACHE_FLUSH             RegisterICacheFlush;
  EFI_EBC_GET_VERSION                       GetVersion;
} EFI_EBC_PROTOCOL;
```

Parameters

CreateThunk	Creates a thunk for an EBC image entry point or protocol service, and returns a pointer to the thunk. See the **CreateThunk()** function description.
UnloadImage	Called when an EBC image is unloaded to allow the interpreter to perform any cleanup associated with the image's execution. See the **UnloadImage()** function description.
RegisterICacheFlush	Called to register a callback function that the EBC interpreter can call to flush the processor instruction cache after creating thunks. See the **RegisterICacheFlush()** function description.
GetVersion	Called to get the version of the associated EBC interpreter. See the **GetVersion()** function description.

Description

The EFI EBC protocol provides services to load and execute EBC images, which will typically be loaded into option ROMs. The image loader will load the EBC image, perform standard relocations, and invoke the `CreateThunk()` service to create a thunk for the EBC image's entry point. The image can then be run using the standard EFI start image services.

EFI_EBC_PROTOCOL.CreateThunk()

Summary

Creates a thunk for an EBC entry point, returning the address of the thunk.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_EBC_CREATE_THUNK) (
  IN  EFI_EBC_PROTOCOL      *This,
  IN  EFI_HANDLE            ImageHandle,
  IN  VOID                  *EbcEntryPoint,
  OUT VOID                  **Thunk
  );
```

Parameters

This	A pointer to the **EFI_EBC_PROTOCOL** instance. This protocol is defined in Section 19.11.
ImageHandle	Handle of image for which the thunk is being created.
EbcEntryPoint	Address of the actual EBC entry point or protocol service the thunk should call.
Thunk	Returned pointer to a thunk created.

Description

A PE32+ EBC image, like any other PE32+ image, contains an optional header that specifies the entry point for image execution. However for EBC images this is the entry point of EBC instructions, so is not directly executable by the native processor. Therefore when an EBC image is loaded, the loader must call this service to get a pointer to native code (thunk) that can be executed which will invoke the interpreter to begin execution at the original EBC entry point.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_INVALID_PARAMETER	Image entry point is not 2-byte aligned.
EFI_OUT_OF_RESOURCES	Memory could not be allocated for the thunk.

EFI_EBC_PROTOCOL.UnloadImage()

Summary

Called prior to unloading an EBC image from memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_EBC_UNLOAD_IMAGE) (
  IN EFI_EBC_PROTOCOL          *This,
  IN EFI_HANDLE                ImageHandle
  );
```

Parameters

This	A pointer to the **EFI_EBC_PROTOCOL** instance. This protocol is defined in Section 19.11.
ImageHandle	Image handle of the EBC image that is being unloaded from memory.

Description

This function is called after an EBC image has exited, but before the image is actually unloaded. It is intended to provide the interpreter with the opportunity to perform any cleanup that may be necessary as a result of loading and executing the image.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_INVALID_PARAMETER	Image handle is not recognized as belonging to an EBC image that has been executed.

EFI_EBC_PROTOCOL.RegisterICacheFlush()

Summary

Registers a callback function that the EBC interpreter calls to flush the processor instruction cache following creation of thunks.

Prototype

```
typedef
EFI_STATUS
(* EFI_EBC_REGISTER_ICACHE_FLUSH) (
  IN EFI_EBC_PROTOCOL                  *This,
  IN EBC_ICACHE_FLUSH                  Flush
  );
```

Parameters

This	A pointer to the **EFI_EBC_PROTOCOL** instance. This protocol is defined in Section 19.11.
Flush	Pointer to a function of type **EBC_ICACH_FLUSH**. See "Related Definitions" below for a detailed description of this type.

Related Definitions

```
typedef
EFI_STATUS
(* EBC_ICACHE_FLUSH) (
  IN EFI_PHYSICAL_ADDRESS         Start,
  IN UINT64                       Length
  );
```

Start	The beginning physical address to flush from the processor's instruction cache.
Length	The number of bytes to flush from the processor's instruction cache.

This is the prototype for the *Flush* callback routine. A pointer to a routine of this type is passed to the EBC **EFI_EBC_REGISTER_ICACHE_FLUSH** protocol service.

Description

An EBC image's original PE32+ entry point is not directly executable by the native processor. Therefore to execute an EBC image, a thunk (which invokes the EBC interpreter for the image's original entry point) must be created for the entry point, and the thunk is executed when the EBC image is started. Since the thunks may be created on-the-fly in memory, the processor's instruction cache may require to be flushed after thunks are created. The caller to this EBC service can provide a pointer to a function to flush the instruction cache for any thunks created after the **`CreateThunk()`** service has been called. If an instruction-cache flush callback is not provided to the interpreter, then the interpreter assumes the system has no instruction cache, or that flushing the cache is not required following creation of thunks.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.

EFI_EBC_PROTOCOL.GetVersion()

Summary

Called to get the version of the interpreter.

Prototype

```
typedef
EFI_STATUS
(* EFI_EBC_GET_VERSION) (
  IN  EFI_EBC_PROTOCOL            *This,
  OUT UINT64                      *Version
  );
```

Parameters

This	A pointer to the **EFI_EBC_PROTOCOL** instance. This protocol is defined in Section 19.11.
Version	Pointer to where to store the returned version of the interpreter.

Description

This function is called to get the version of the loaded EBC interpreter. The value and format of the returned version is identical to that returned by the EBC BREAK 1 instruction.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_INVALID_PARAMETER	Version pointer is NULL.

19.12 EBC Tools

19.12.1 EBC C Compiler

This section describes the responsibilities of the EBC C compiler. To fully specify these responsibilities requires that the thunking mechanisms between EBC and native code be described.

19.12.2 C Coding Convention

The EBC C compiler supports only the C programming language. There is no support for C++, inline assembly, floating point types/operations, or C calling conventions other than CDECL.

Pointer type in C is supported only as 64-bit pointer. The code should be 64-bit pointer ready (not assign pointers to integers and vice versa).

The compiler does not support user-defined sections through pragmas.

Global variables containing pointers that are initialized will be put in the uninitialized VarBss section and the compiler will generate code to initialize these variables during load time. The code will be placed in an init text section. This compiler-generated code will be executed before the actual image entry point is executed.

19.12.3 EBC Interface Assembly Instructions

The EBC instruction set includes two forms of a CALL instruction that can be used to invoke external protocols. Their assembly language formats are:

CALLEX Immed64

CALLEX32 {@}R$_1$ {Immed32}

Both forms can be used to invoke external protocols at an absolute address specified by the immediate data and/or register operand. The second form also supports jumping to code at a relative address. When one of these instructions is executed, the interpreter is responsible for thunking arguments and then jumping to the destination address. When the called function returns, code begins execution at the EBC instruction following the CALL instruction. The process by which this happens is called thunking. Later sections describe this operation in detail.

19.12.4 Stack Maintenance and Argument Passing

There are several EBC assembly instructions that directly manipulate the stack contents and stack pointer. These instructions operate on the EBC stack, not the interpreter stack. The instructions include the EBC PUSH, POP, PUSHn, and POPn, and all forms of the MOV instructions.

These instructions must adjust the EBC stack pointer in the same manner as equivalent instructions of the native instruction set. With this implementation, parameters pushed on the stack by an EBC driver can be accessed normally for stack-based native code. If native code expects parameters in registers, then the interpreter thunking process must transfer the arguments from EBC stack to the appropriate processor registers. The process would need to be reversed when native code calls EBC.

19.12.5 Native to EBC Arguments Calling Convention

The calling convention for arguments passed to EBC functions follows the standard CDECL calling convention. The arguments must be pushed as their native size. After the function arguments have been pushed on the stack, execution is passed to the called EBC function. The overhead of thunking the function parameters depends on the standard parameter passing convention for the host processor. The implementation of this functionality is left to the interpreter.

19.12.6 EBC to Native Arguments Calling Convention

When EBC makes function calls via function pointers, the EBC C compiler cannot determine whether the calls are to native code or EBC. It therefore assumes that the calls are to native code, and emits the appropriate EBC CALLEX instructions. To be compatible with calls to native code, the calling convention of EBC calling native code must follow the parameter passing convention of the native processor. The EBC C compiler generates EBC instructions that push all arguments on the stack. The interpreter is then responsible for performing the necessary thunking. The exact implementation of this functionality is left to the interpreter.

19.12.7 EBC to EBC Arguments Calling Convention

If the EBC C compiler is able to determine that a function call is to a local function, it can emit a standard EBC CALL instruction. In this case, the function arguments are passed as described in the other sections of this specification.

19.12.8 Function Returns

When EBC calls an external function, the thunking process includes setting up the host processor stack or registers such that when the called function returns, execution is passed back to the EBC at the instruction following the call. The implementation is left to the interpreter, but it must follow the standard function return process of the host processor. Typically this will require the interpreter to push the return address on the stack or move it to a processor register prior to calling the external function.

19.12.9 Function Return Values

EBC function return values of 8 bytes or less are returned in VM general-purpose register **R7**. Returning values larger than 8 bytes on the stack is not supported. Instead, the caller or callee must allocate memory for the return value, and the caller can pass a pointer to the callee, or the callee can return a pointer to the value in the standard return register **R7**.

If an EBC function returns to native code, then the interpreter thunking process is responsible for transferring the contents of **R7** to an appropriate location such that the caller has access to the value using standard native code. Typically the value will be transferred to a processor register. Conversely, if a native function returns to an EBC function, the interpreter is responsible for transferring the return value from the native return memory or register location into VM register **R7**.

19.12.10 Thunking

Thunking is the process by which transitions between execution of native and EBC are handled. The major issues that must be addressed for thunking are the handling of function arguments, how the external function is invoked, and how return values and function returns are handled. The following sections describe the thunking process for the possible transitions.

19.12.10.1 Thunking EBC to Native Code

By definition, all external calls from within EBC are calls to native code. The EBC CALLEX instructions are used to make these calls. A typical application for EBC calling native code would be a simple “Hello World” driver. For an EFI driver, the code could be written as shown below.

```
EFI_STATUS EfiMain (
    IN EFI_HANDLE            ImageHandle,
    IN EFI_SYSTEM_TABLE      *ST
    )
{
   ST->ConOut->OutputString(ST->ConOut, L”Hello World!”);
   return EFI_SUCCESS;
}
```

This C code, when compiled to EBC assembly, could result in two PUSHn instructions to push the parameters on the stack, some code to get the absolute address of the `OutputString()` function, then a CALLEX instruction to jump to native code. Typical pseudo assembly code for the function call could be something like the following:

```
PUSHn       _HelloString
PUSHn       _ConOut
MOVnw       R1, _OutputString
CALLEX64    R1
```

The interpreter is responsible for executing the PUSHn instructions to push the arguments on the EBC stack when interpreting the PUSHn instructions. When the CALLEX instruction is encountered, it must thunk to external native code. The exact thunking mechanism is native processor dependent. For example, an IA-32 thunking implementation could simply move the system stack pointer to point to the EBC stack, then perform a CALL to the absolute address specified in VM register **R1**. However, the function calling convention for the Itanium processor family calls for the first 8 function arguments being passed in registers. Therefore, the Itanium processor family thunking mechanism requires the arguments to be copied from the EBC stack into processor registers. Then a CALL can be performed to jump to the absolute address in VM register **R1**. Note that since the interpreter is not aware of the number of arguments to the function being called, the maximum amount of data may be copied from the EBC stack into processor registers.

19.12.10.2 Thunking Native Code to EBC

An EBC driver may install protocols for use by other EBC drivers or EFI drivers or applications. These protocols provide the mechanism by which external native code can call EBC. Typical EFI C code to install a generic protocol is shown below.

```
EFI_STATUS Foo(UINT32 Arg1, UINT32 Arg2);

MyProtInterface->Service1= Foo;

Status = LibInstallProtocolInterfaces (&Handle, &MyProtGUID,
MyProtInterface, NULL);
```

To support thunking native code to EBC, the EBC compiler resolves all EBC function pointers using one level of indirection. In this way, the address of an EBC function actually becomes the address of a piece of native (thunk) code that invokes the interpreter to execute the actual EBC function. As a result of this implementation, any time the address of an EBC function is taken, the EBC C compiler must generate the following:

- A 64-bit function pointer data object that contains the actual address of the EBC function
- EBC initialization code that is executed before the image entry point that will execute EBC BREAK 5 instructions to create thunks for each function pointer data object
- Associated relocations for the above

So for the above code sample, the compiler must generate EBC initialization code similar to the following. This code is executed prior to execution of the actual EBC driver's entry point.

```
MOVqq  R7, Foo_pointer    ; get address of Foo pointer
BREAK  5                  ; create a thunk for the function
```

The BREAK instruction causes the interpreter to create native thunk code elsewhere in memory, and then modify the memory location pointed to by R7 to point to the newly created thunk code for EBC function Foo. From within EBC, when the address of Foo is taken, the address of the thunk is actually returned. So for the assignment of the protocol Service1 above, the EBC C compiler will generate something like the following:

```
MOVqq  R7, Foo_pointer    ; get address of Foo function pointer
MOVqq  R7, @R7            ; one level of indirection
MOVn   R6, _MyProtInterface->Service1 ; get address of variable
MOVqq  @R6, R7            ; address of thunk to ->Service1
```

19.12.10.3 Thunking EBC to EBC

EBC can call EBC via function pointers or protocols. These two mechanisms are treated identically by the EBC C compiler, and are performed using EBC CALLEX instructions. For EBC to call EBC, the EBC being called must have provided the address of the function. As described above, the address is actually the address of native thunk code for the actual EBC function. Therefore, when EBC calls EBC, the interpreter assumes native code is being called so prepares function arguments accordingly, and then makes the call. The native thunk code assumes native code is calling EBC, so will basically “undo” the preparation of function arguments, and then invoke the interpreter to execute the actual EBC function of interest.

19.12.11 EBC Linker

New constants must be defined for use by the linker in processing EBC images. For EBC images, the linker must set the machine type in the PE file header accordingly to indicate that the image contains EBC.

```
#define IMAGE_FILE_MACHINE_EBC       0x0EBC
```

In addition, the linker must support EBC images with of the following subsystem types as set in a PE32+ optional header:

```
#define IMAGE_SUBSYSTEM_EFI_APPLICATION                10
#define IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER        11
#define IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER             12
```

For EFI EBC images and object files, the following relocation types must be supported:

```
// No relocations required
#define IMAGE_REL_EBC_ABSOLUTE       0x0000
// 32-bit address w/o image base
#define IMAGE_REL_EBC_ADDR32NB       0x0001
// 32-bit relative address from byte following relocs
#define IMAGE_REL_EBC_REL32          0x0002
// Section table index
#define IMAGE_REL_EBC_SECTION        0x0003
// Offset within section
#define IMAGE_REL_EBC_SECREL         0x0004
```

The ADDR32NB relocation is used internally to the linker when RVAs are emitted. It also is used for version resources which probably will not be used. The REL32 relocation is for PC relative addressing on code. The SECTION and SECREL relocations are used for debug information.

19.12.12 Image Loader

The EFI image loader is responsible for loading an executable image into memory and performing any fixups prior to execution of the image. For EBC images, the image loader must also invoke the interpreter protocol to create a thunk for the image entry point and return the address of this thunk. After loading the image in this manner, the image can be executed in the standard manner. To implement this functionality, only minor changes will be made to EFI service **LoadImage()**, and no changes should be made to **StartImage()**.

After the image is unloaded, the EFI image load service must call the EBC **UnloadImage()** service to perform any cleanup to complete unloading of the image. Typically this will include freeing up any memory allocated for thunks for the image during load and execution.

19.12.13 Debug Support

The interpreter must support debugging in an EFI environment per the EFI debug support protocol.

19.13 VM Exception Handling

This section lists the different types of exceptions that the VM may assert during execution of an EBC image. If a debugger is attached to the EBC driver via the EFI debug support protocol, then the debugger should be able to capture and identify the exception type. If a debugger is not attached, then depending on the severity of the exception, the interpreter may do one of the following:

- Invoke the EFI ASSERT() macro, which will typically display an error message and halt the system
- Sit in a while(1) loop to hang the system
- Ignore the exception and continue execution of the image (minor exceptions only)

It is a platform policy decision as to the action taken in response to EBC exceptions. The following sections describe the exceptions that may be generated by the VM.

19.13.1 Divide By 0 Exception

A divide-by-0 exception can occur for the EBC instructions DIV, DIVU, MOD, and MODU.

19.13.2 Debug Break Exception

A debug break exception occurs if the VM encounters a BREAK instruction with a break code of 3.

19.13.3 Invalid Opcode Exception

An invalid opcode exception will occur if the interpreter encounters a reserved opcode during execution.

19.13.4 Stack Fault Exception

A stack fault exception can occur if the interpreter detects that function nesting within the interpreter or system interrupts was sufficient to potentially corrupt the EBC image's stack contents. This exception could also occur if the EBC driver attempts to adjust the stack pointer outside the range allocated to the driver.

19.13.5 Alignment Exception

An alignment exception can occur if the particular implementation of the interpreter does not support unaligned accesses to data or code. It may also occur if the stack pointer or instruction pointer becomes misaligned.

19.13.6 Instruction Encoding Exception

An instruction encoding exception can occur for the following:

- For some instructions, if an Operand 1 index is specified and Operand 1 is direct
- If an instruction encoding has reserved bits set to values other than 0
- If an instruction encoding has a field set to a reserved value.

19.13.7 Bad Break Exception

A bad break exception occurs if the VM encounters a BREAK instruction with a break code of 0, or any other unrecognized or unsupported break code.

19.13.8 Undefined Exception

An undefined exception can occur for other conditions detected by the VM. The cause of such an exception is dependent on the VM implementation, but will most likely include internal VM faults.

19.14 Option ROM Formats

The new option ROM capability is designed to be a departure from the legacy method of formatting an option ROM. PCI local bus add-in cards are the primary targets for this design although support for future bus types will be added as necessary. EFI EBC drivers can be stored in option ROMs or on hard drives in an EFI system partition.

The new format defined for the EFI specification is intended to coexist with legacy format PCI Expansion ROM images. This provides the ability for IHVs to make a single option ROM binary that contains both legacy and new format images at the same time. This is important for the ability to have single add-in card SKUs that can work in a variety of systems both with and without native support for EFI. Support for multiple image types in this way provides a smooth migration path during the period before widespread adoption of EFI drivers as the primary means of support for software needed to accomplish add-in card operation in the pre-OS boot timeframe.

19.14.1 EFI Drivers for PCI Add-in Cards

The location mechanism for EFI drivers in PCI option ROM containers is described fully in Chapter 12 (section 12.4.2). Readers should refer to this section for complete details of the scheme and associated data structures.

19.14.2 Non-PCI Bus Support

EFI expansion ROMs are not supported on any other bus besides PCI local bus in the current revision of the EFI specification.

This means that support for EFI drivers in legacy ISA add-in card ROMs is explicitly excluded.

Support for EFI drivers to be located on add-in card type devices for future bus designs other than PCI local bus will be added to future revisions of the EFI specification. This support will depend upon the specifications that govern such new bus designs with respect to the mechanisms defined for support of driver code on devices.

Appendix A
GUID and Time Formats

All EFI GUIDs (Globally Unique Identifiers) have the format described in Appendix J of the *Wired for Management Baseline Specification*. This document references the format of the GUID, but implementers must reference the Wired for Management specifications for algorithms to generate GUIDs. The following table defines the format of an EFI GUID (128 bits).

Table A-1. EFI GUID Format

Mnemonic	Byte Offset	Byte Length	Description
TimeLow	0	4	The low field of the timestamp.
TimeMid	4	2	The middle field of the timestamp.
TimeHighAndVersion	6	2	The high field of the timestamp multiplexed with the version number.
ClockSeqHighAndReserved	8	1	The high field of the clock sequence multiplexed with the variant.
ClockSeqLow	9	1	The low field of the clock sequence.
Node	10	6	The spatially unique node identifier. This can be based on any IEEE 802 address obtained from a network card. If no network card exists in the system, a cryptographic-quality random number can be used.

All EFI time is stored in the format described by Appendix J of the *Wired for Management Baseline Specification*. This appendix for GUID defines a 60-bit timestamp format that is used to generate the GUID. All EFI time information is stored in 64-bit structures that contain the following format: The timestamp is a 60-bit value that is represented by Coordinated Universal Time (UTC) as a count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582 (the date of Gregorian reform to the Christian calendar). This time value will not roll over until the year 3400 AD. It is assumed that a future version of the EFI specification can deal with the year-3400 issue by extending this format if necessary.

Appendix B Console

The EFI console was designed so that it could map to common console devices. This appendix explains how an EFI console could map to a VGA with PC AT 101/102, PC ANSI, or ANSI X3.64 consoles.

B.1 SIMPLE_INPUT

Table B-1 gives examples of how an EFI scan code can be mapped to ANSI X3.64 terminal, PCANSI terminal, or an AT 101/102 keyboard. PC ANSI terminals support an escape sequence that begins with the ASCII character 0x1b and is followed by the ASCII character 0x5B, " [". ASCII characters that define the control sequence that should be taken follow the escape sequence. (The escape sequence does not contain spaces, but spaces are used in Table B-1 to ease the reading of the table.) ANSI X3.64, when combined with ISO 6429, can be used to represent the same subset of console support required by EFI. ANSI X3.64 uses a single character escape sequence CSI: ASCII character 0x9B. ANSI X3.64 can optionally use the same two-character escape sequence "ESC [". ANSI X3.64 and ISO 6429 support the same escape codes as PC ANSI.

Table B-1. EFI Scan Codes for `SIMPLE_INPUT`

EFI Scan Code	Description	ANSI X3.64 Codes	PC ANSI Codes	AT 101/102 Keyboard Scan Codes
0x00	Null scan code	N/A	N/A	N/A
0x01	Move cursor up 1 row	CSI A	ESC [A	0xe0, 0x48
0x02	Move cursor down 1 row	CSI B	ESC [B	0xe0, 0x50
0x03	Move cursor right 1 column	CSI C	ESC [C	0xe0, 0x4d
0x04	Move cursor left 1 column	CSI D	ESC [D	0xe0, 0x4b
0x05	Home	CSI H	ESC [H	0xe0, 0x47
0x06	End	CSI K	ESC [K	0xe0, 0x4f
0x07	Insert	CSI @	ESC [@	0xe0, 0x52
0x08	Delete	CSI P	ESC [P	0xe0, 0x53
0x09	Page Up	CSI ?	ESC [?	0xe0, 0x49
0x0a	Page Down	CSI /	ESC [/	0xe0, 0x51

continued

Table B-1. EFI Scan Codes for SIMPLE_INPUT (continued)

EFI Scan Code	Description	ANSI X3.64 Codes	PC ANSI Codes	AT 101/102 Keyboard Scan Codes
0x0b	Function 1	CSI O P	ESC [O P	0x3b
0x0c	Function 2	CSI O Q	ESC [O Q	0x3c
0x0d	Function 3	CSI O w	ESC [O w	0x3d
0x0e	Function 4	CSI O x	ESC [O x	0x3e
0x0f	Function 5	CSI O t	ESC [O t	0x3f
0x10	Function 6	CSI O u	ESC [O u	0x40
0x11	Function 7	CSI O q	ESC [O q	0x41
0x12	Function 8	CSI O r	ESC [O r	0x42
0x13	Function 9	CSI O p	ESC [O p	0x43
0x14	Function 10	CSI O M	ESC [O M	0x44
0x17	Escape	CSI	ESC	0x01

B.2 SIMPLE_TEXT_OUTPUT

Table B-2 defines how the programmatic methods of the **SIMPLE_TEXT_OUPUT** protocol could be implemented as PC ANSI or ANSI X3.64 terminals. Detailed descriptions of PC ANSI and ANSI X3.64 escape sequences are as follows. The same type of operations can be supported via a PC AT type INT 10h interface.

Table B-2. Control Sequences That Can Be Used to Implement SIMPLE_TEXT_OUTPUT

PC ANSI Codes	ANSI X3.64 Codes	Description
ESC [2 J	CSI 2 J	Clear Display Screen.
ESC [0 m	CSI 0 m	Normal Text.
ESC [1 m	CSI 1 m	Bright Text.
ESC [7 m	CSI 7 m	Reversed Text.
ESC [30 m	CSI 30 m	Black foreground, compliant with ISO Standard 6429.
ESC [31 m	CSI 31 m	Red foreground, compliant with ISO Standard 6429.
ESC [32 m	CSI 32 m	Green foreground, compliant with ISO Standard 6429.
ESC [33 m	CSI 33 m	Yellow foreground, compliant with ISO Standard 6429.
ESC [34 m	CSI 34 m	Blue foreground, compliant with ISO Standard 6429.

continued

Table B-2. Control Sequences That Can Be Used to Implement `SIMPLE_TEXT_OUTPUT` (continued)

PC ANSI Codes	ANSI X3.64 Codes	Description
ESC [35 m	CSI 35 m	Magenta foreground, compliant with ISO Standard 6429.
ESC [36 m	CSI 36 m	Cyan foreground, compliant with ISO Standard 6429.
ESC [37 m	CSI 37 m	White foreground, compliant with ISO Standard 6429.
ESC [40 m	CSI 40 m	Black background, compliant with ISO Standard 6429.
ESC [41 m	CSI 41 m	Red background, compliant with ISO Standard 6429.
ESC [42 m	CSI 42 m	Green background, compliant with ISO Standard 6429.
ESC [43 m	CSI 43 m	Yellow background, compliant with ISO Standard 6429.
ESC [44 m	CSI 44 m	Blue background, compliant with ISO Standard 6429.
ESC [45 m	CSI 45 m	Magenta background, compliant with ISO Standard 6429.
ESC [46 m	CSI 46 m	Cyan background, compliant with ISO Standard 6429.
ESC [47 m	CSI 47 m	White background, compliant with ISO Standard 6429.
ESC [3 h	CSI = 3 h	Set Mode 80x25 color.
ESC [*row;col* H	CSI *row;col* H	Set cursor position to *row;col*. *Row* and *col* are strings of ASCII digits.

Appendix C
Device Path Examples

This appendix presents an example EFI Device Path and explains its relationship to the ACPI name space. An example system design is presented along with its corresponding ACPI name space. These physical examples are mapped back to EFI Device Paths.

C.1 Example Computer System

Figure C-1 represents a hypothetical computer system architecture that will be used to discuss the construction of EFI Device Paths. The system consists of a memory controller that connects directly to the processors' front side bus. The memory controller is only part of a larger chipset, and it connects to a root PCI host bridge chip, and a secondary root PCI host bridge chip. The secondary PCI host bridge chip produces a PCI bus that contains a PCI to PCI bridge. The root PCI host bridge produces a PCI bus, and also contains USB, ATA66, and AC '97 controllers. The root PCI host bridge also contains an LPC bus that is used to connect a SIO (Super IO) device. The SIO contains a PC-AT-compatible floppy disk controller, and other PC-AT-compatible devices like a keyboard controller.

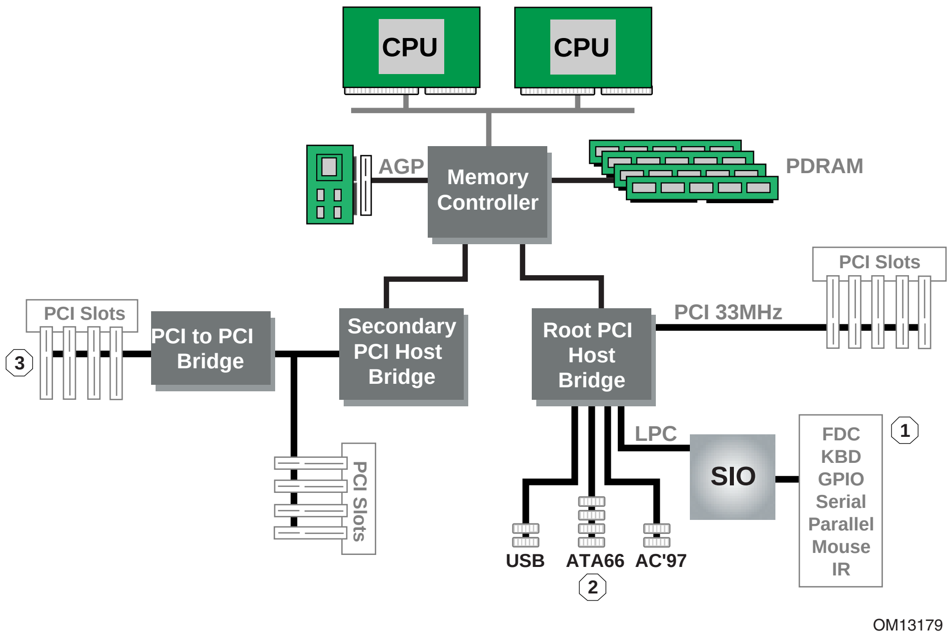


Figure C-1. Example Computer System

The remainder of this appendix describes how to construct a device path for three example devices from the system in Figure C-1. The following is a list of the examples used:

- Legacy floppy
- IDE Disk
- Secondary root PCI bus with PCI to PCI bridge

Figure C-2 is a partial ACPI name space for the system in Figure C-1. Figure C-2 is based on Figure 5-3 in the *Advanced Configuration and Power Interface Specification*.

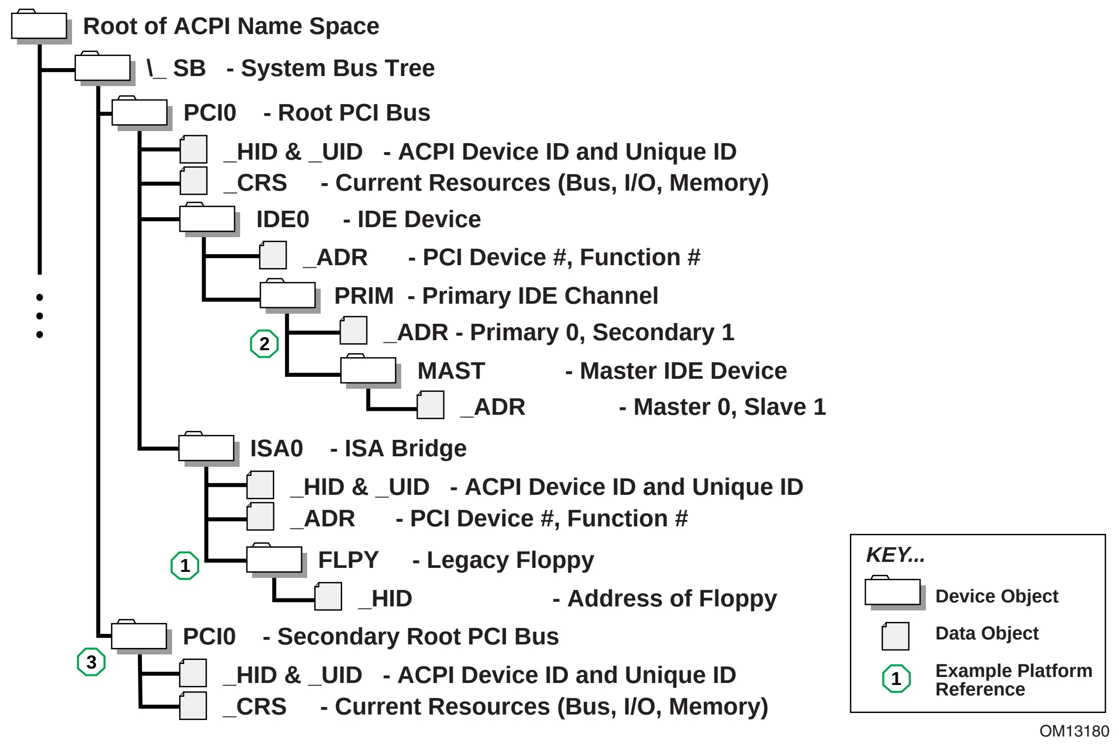

Root of ACPI Name Space
_ SB - System Bus Tree
PCI0 - Root PCI Bus
_HID & _UID - ACPI Device ID and Unique ID
_CRS - Current Resources (Bus, I/O, Memory)
IDE0 - IDE Device
_ADR - PCI Device #, Function #
PRIM - Primary IDE Channel
2
_ADR - Primary 0, Secondary 1
MAST - Master IDE Device
_ADR - Master 0, Slave 1
ISA0 - ISA Bridge
_HID & _UID - ACPI Device ID and Unique ID
_ADR - PCI Device #, Function #
1
FLPY - Legacy Floppy
_HID - Address of Floppy
PCI0 - Secondary Root PCI Bus
3
_HID & _UID - ACPI Device ID and Unique ID
_CRS - Current Resources (Bus, I/O, Memory)
KEY...
Device Object
Data Object
1
Example Platform Reference
OM13180

Figure C-2. Partial ACPI Name Space for Example System

C.2 Legacy Floppy

The legacy floppy controller is contained in the SIO chip that is connected root PCI bus host bridge chip. The root PCI host bridge chip produces PCI bus 0, and other resources that appear directly to the processors in the system.

In ACPI this configuration is represented in the _SB, system bus tree, of the ACPI name space. PCI0 is a child of _SB and it represents the root PCI host bridge. The SIO appears to the system to be a set of ISA devices, so it is represented as a child of PCI0 with the name ISA0. The floppy controller is represented by FLPY as a child of the ISA0 bus.

The EFI Device Path for the legacy floppy is defined in Table C-1. It would contain entries for the following things:

- Root PCI Bridge. ACPI Device Path _HID PNP0A03, _UID 0. ACPI name space _SB\PCI0
- PCI to ISA Bridge. PCI Device Path with device and function of the PCI to ISA bridge. ACPI name space _SB\PCI0\ISA0
- Floppy Plug and Play ID. ACPI Device Path _HID PNP0303, _UID 0. ACPI name space _SB\PCI0\ISA0\FLPY
- End Device Path

Table C-1. Legacy Floppy Device Path

Byte Offset	Byte Length	Data	Description
0	1	0x02	**Generic Device Path Header** – Type ACPI Device Path
1	1	0x01	Sub type – ACPI Device Path
2	2	0x0C	Length
4	4	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes
8	4	0x0000	_UID
C	1	0x01	**Generic Device Path Header** – Type Hardware Device Path
D	1	0x01	Sub type PCI Device Path
E	2	0x06	Length
10	1	0x00	PCI Function
11	1	0x10	PCI Device
12	1	0x02	**Generic Device Path Header** – Type ACPI Device Path
13	1	0x01	Sub type – ACPI Device Path
14	2	0x0C	Length
16	4	0x41D0, 0x0303	_HID PNP0303
1A	4	0x0000	_UID
1E	1	0xFF	**Generic Device Path Header** – Type End Device Path
1F	1	0xFF	Sub type – End Device Path
20	2	0x04	Length

C.3 IDE Disk

The IDE Disk controller is a PCI device that is contained in a function of the root PCI host bridge. The root PCI host bridge is a multi function device and has a separate function for chipset registers, USB, and IDE. The disk connected to the IDE ATA bus is defined as being on the primary or secondary ATA bus, and of being the master or slave device on that bus.

In ACPI this configuration is represented in the _SB, system bus tree, of the ACPI name space. PCI0 is a child of _SB and it represents the root PCI host bridge. The IDE controller appears to the system to be a PCI device with some legacy properties, so it is represented as a child of PCI0 with the name IDE0. PRIM is a child of IDE0 and it represents the primary ATA bus of the IDE controller. MAST is a child of PRIM and it represents that this device is the ATA master device on this primary ATA bus.

The EFI Device Path for the PCI IDE controller is defined in Table C-2. It would contain entries for the following things:

- Root PCI Bridge. ACPI Device Path _HID PNP0A03, _UID 0. ACPI name space _SB\PCI0
- PCI IDE controller. PCI Device Path with device and function of the IDE controller. ACPI name space _SB\PCI0\IDE0
- ATA Address. ATA Messaging Device Path for Primary bus and Master device. ACPI name space _SB\PCI0\IDE0\PRIM\MAST
- End Device Path

Table C-2. IDE Disk Device Path

Byte Offset	Byte Length	Data	Description
0	1	0x02	**Generic Device Path Header** – Type ACPI Device Path
1	1	0x01	Sub type – ACPI Device Path
2	2	0x0C	Length
4	4	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes
8	4	0x0000	_UID
C	1	0x01	**Generic Device Path Header** – Type Hardware Device Path
D	1	0x01	Sub type PCI Device Path
E	2	0x06	Length
10	1	0x01	PCI Function
11	1	0x10	PCI Device
12	1	0x03	**Generic Device Path Header** – Messaging Device Path
13	1	0x01	Sub type – ATAPI Device Path
14	2	0x06	Length
16	1	0x00	Primary =0, Secondary = 1
17	1	0x00	Master = 0, Slave = 1
18	2	0x0000	LUN
1A	1	0xFF	**Generic Device Path Header** – Type End Device Path
1B	1	0xFF	Sub type – End Device Path
1C	2	0x04	Length

C.4 Secondary Root PCI Bus with PCI to PCI Bridge

The secondary PCI host bridge materializes a second set of PCI buses into the system. The PCI buses on the secondary PCI host bridge are totally independent of the PCI buses on the root PCI host bridge. The only relationship between the two is they must be configured to not consume the same resources. The primary PCI bus of the secondary PCI host bridge also contains a PCI to PCI bridge. There is some arbitrary PCI device plugged in behind the PCI to PCI bridge in a PCI slot.

In ACPI this configuration is represented in the _SB, system bus tree, of the ACPI name space. PCI1 is a child of _SB and it represents the secondary PCI host bridge. The PCI to PCI bridge and the device plugged into the slot on its primary bus are not described in the ACPI name space. These devices can be fully configured by following the applicable PCI specification.

The EFI Device Path for the secondary root PCI bridge with a PCI to PCI bridge is defined in Table C-3. It would contain entries for the following things:

- Root PCI Bridge. ACPI Device Path _HID PNP0A03, _UID 1. ACPI name space _SB\PCI1
- PCI to PCI Bridge. PCI Device Path with device and function of the PCI Bridge. ACPI name space _SB\PCI1, PCI to PCI bridges are defined by PCI specification and not ACPI.
- PCI Device. PCI Device Path with the device and function of the PCI device. ACPI name space _SB\PCI1, PCI devices are defined by PCI specification and not ACPI.
- End Device Path.

Table C-3. Secondary Root PCI Bus with PCI to PCI Bridge Device Path

Byte Offset	Byte Length	Data	Description
0	1	0x02	**Generic Device Path Header** – Type ACPI Device Path
1	1	0x01	Sub type – ACPI Device Path
2	2	0x0C	Length
4	4	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes
8	4	0x0001	_UID
C	1	0x01	**Generic Device Path Header** – Type Hardware Device Path
D	1	0x01	Sub type PCI Device Path
E	2	0x06	Length
10	1	0x00	PCI Function for PCI to PCI bridge
11	1	0x0c	PCI Device for PCI to PCI bridge
12	1	0x01	**Generic Device Path Header** – Type Hardware Device Path
13	1	0x01	Sub type PCI Device Path
14	2	0x08	Length
16	1	0x00	PCI Function for PCI Device
17	1	0x00	PCI Device for PCI Device
18	1	0xFF	**Generic Device Path Header** – Type End Device Path
19	1	0xFF	Sub type – End Device Path
1A	2	0x04	Length

C.5 ACPI Terms

Names in the ACPI name space that start with an underscore ("_") are reserved by the ACPI specification and have architectural meaning. All ACPI names in the name space are four characters in length. The following four ACPI names are used in this specification.

_ADR. The Address on a bus that has standard enumeration. An example would be PCI, where the enumeration method is described in the PCI Local Bus specification.

_CRS. The current resource setting of a device. A _CRS is required for devices that are not enumerated in a standard fashion. _CRS is how ACPI converts nonstandard devices into Plug and Play devices.

_HID. Represents a device's Plug and Play hardware ID, stored as a 32-bit compressed EISA ID. _HID objects are optional in ACPI. However, a _HID object must be used to describe any device that will be enumerated by the ACPI driver in the OS. This is how ACPI deals with non–Plug and Play devices.

_UID. Is a serial number style ID that does not change across reboots. If a system contains more than one device that reports the same _HID, each device must have a unique _UID. The _UID only needs to be unique for device that have the exact same _HID value.

C.6 EFI Device Path as a Name Space

Figure C-3 shows the EFI Device Path for the example system represented as a name space. The Device Path can be represented as a name space, but EFI does support manipulating the Device Path as a name space. You can only access Device Path information by locating the **DEVICE_PATH_INTERFACE** from a handle. Not all the nodes in a Device Path will have a handle.

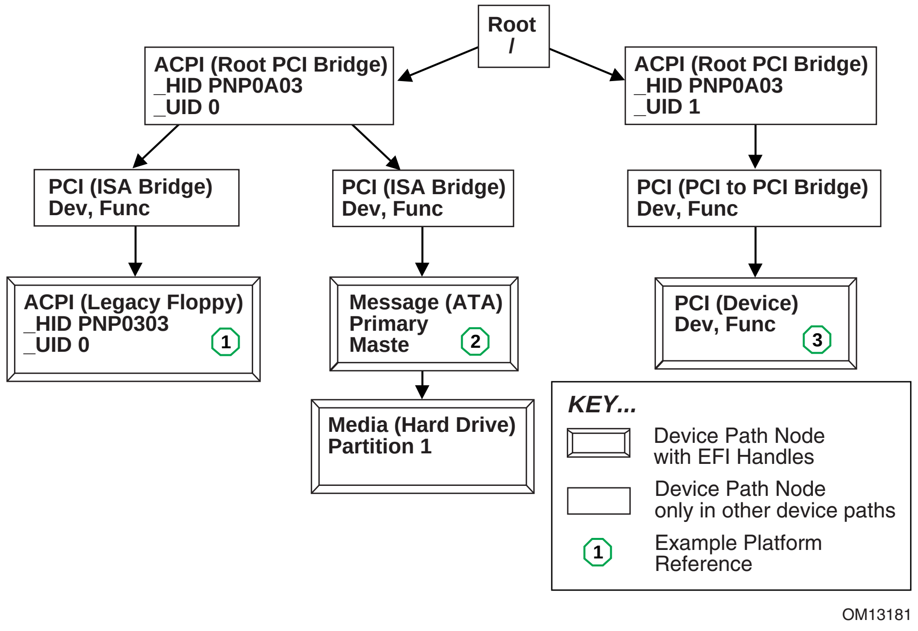


Figure C-3. EFI Device Path Displayed As a Name Space

Appendix D
Status Codes

EFI interfaces return an **EFI_STATUS** code. Table D-2, Table D-3, and Table D-4 list these codes for success, errors, and warnings, respectively. Error codes also have their highest bit set, so all error codes have negative values. The range of status codes that have the highest bit set and the next to highest bit clear are reserved for use by EFI. The range of status codes that have both the highest bit set and the next to highest bit set are reserved for use by OEMs. Success and warning codes have their highest bit clear, so all success and warning codes have positive values. The range of status codes that have both the highest bit clear and the next to highest bit clear are reserved for use by EFI. The range of status code that have the highest bit clear and the next to highest bit set are reserved for use by OEMs. Table D-1 lists the status code ranges described above.

Table D-1. EFI_STATUS Codes Ranges

IA-32 Range	Itanium Architecture Range	Description
`0x00000000-` `0x3fffffff`	`0x0000000000000000-` `0x3fffffffffffffff`	Success and warning codes reserved for use by EFI. See Table D-2 and Table D-4 for valid values in this range.
`0x40000000-` `0x7fffffff`	`0x4000000000000000-` `0x7fffffffffffffff`	Success and warning codes reserved for use by OEMs.
`0x80000000-` `0xbfffffff`	`0x8000000000000000-` `0xbfffffffffffffff`	Error codes reserved for use by EFI. See Table D-3 for valid values for this range.
`0xc0000000-` `0xffffffff`	`0xc000000000000000-` `0xffffffffffffffff`	Error codes reserved for use by OEMs.

Table D-2. EFI_STATUS Success Codes (High Bit Clear)

Mnemonic	Value	Description
EFI_SUCCESS	0	The operation completed successfully.

Table D-3. EFI_STATUS Error Codes (High Bit Set)

Mnemonic	Value	Description
EFI_LOAD_ERROR	1	The image failed to load.
EFI_INVALID_PARAMETER	2	A parameter was incorrect.
EFI_UNSUPPORTED	3	The operation is not supported.
EFI_BAD_BUFFER_SIZE	4	The buffer was not the proper size for the request.
EFI_BUFFER_TOO_SMALL	5	The buffer is not large enough to hold the requested data. The required buffer size is returned in the appropriate parameter when this error occurs.

continued

Table D-3. EFI_STATUS Error Codes (High Bit Set) (continued)

Mnemonic	Value	Description
EFI_NOT_READY	6	There is no data pending upon return.
EFI_DEVICE_ERROR	7	The physical device reported an error while attempting the operation.
EFI_WRITE_PROTECTED	8	The device cannot be written to.
EFI_OUT_OF_RESOURCES	9	A resource has run out.
EFI_VOLUME_CORRUPTED	10	An inconstancy was detected on the file system causing the operating to fail.
EFI_VOLUME_FULL	11	There is no more space on the file system.
EFI_NO_MEDIA	12	The device does not contain any medium to perform the operation.
EFI_MEDIA_CHANGED	13	The medium in the device has changed since the last access.
EFI_NOT_FOUND	14	The item was not found.
EFI_ACCESS_DENIED	15	Access was denied.
EFI_NO_RESPONSE	16	The server was not found or did not respond to the request.
EFI_NO_MAPPING	17	A mapping to a device does not exist.
EFI_TIMEOUT	18	The timeout time expired.
EFI_NOT_STARTED	19	The protocol has not been started.
EFI_ALREADY_STARTED	20	The protocol has already been started.
EFI_ABORTED	21	The operation was aborted.
EFI_ICMP_ERROR	22	An ICMP error occurred during the network operation.
EFI_TFTP_ERROR	23	A TFTP error occurred during the network operation.
EFI_PROTOCOL_ERROR	24	A protocol error occurred during the network operation.
EFI_INCOMPATIBLE_VERSION	25	The function encountered an internal version that was incompatible with a version requested by the caller.
EFI_SECURITY_VIOLATION	26	The function was not performed due to a security violation.
EFI_CRC_ERROR	27	A CRC error was detected.

Table D-4. EFI_STATUS Warning Codes (High Bit Clear)

Mnemonic	Value	Description
EFI_WARN_UNKOWN_GLYPH	1	The Unicode string contained one or more characters that the device could not render and were skipped.
EFI_WARN_DELETE_FAILURE	2	The handle was closed, but the file was not deleted.
EFI_WARN_WRITE_FAILURE	3	The handle was closed, but the data to the file was not flushed properly.
EFI_WARN_BUFFER_TOO_SMALL	4	The resulting buffer was too small, and the data was truncated to the buffer size.

Appendix E 32/64-Bit UNDI Specification

E.1 Introduction

This appendix defines the 32/64-bit H/W and S/W Universal Network Driver Interfaces (UNDIs). These interfaces provide one method for writing a network driver; other implementations are possible.

NOTE

This is the Beta-1 version of the 32/64-bit UNDI Specification.

E.1.1 Definitions

Table E-1. Definitions

Term	Definition
BC	**BaseCode** The PXE BaseCode, included as a core protocol in EFI, is comprised of a simple network stack (UDP/IP) and a few common network protocols (DHCP, Bootserver Discovery, TFTP) that are useful for remote booting machines.
LOM	**LAN On Motherboard** This is a network device that is built onto the motherboard (or baseboard) of the machine.
NBP	**Network Bootstrap Program** This is the first program that is downloaded into a machine that has selected a PXE capable device for remote boot services. A typical NBP examines the machine it is running on to try to determine if the machine is capable of running the next layer (OS or application). If the machine is not capable of running the next layer, control is returned to the EFI boot manager and the next boot device is selected. If the machine is capable, the next layer is downloaded and control can then be passed to the downloaded program. Though most NBPs are OS loaders, NBPs can be written to be standalone applications such as diagnostics, backup/restore, remote management agents, browsers, etc.
NIC	**Network Interface Card** Technically, this is a network device that is inserted into a bus on the motherboard or in an expansion board. For the purposes of this document, the term NIC will be used in a generic sense, meaning any device that enables a network connection (including LOMs and network devices on external busses (USB, 1394, etc.)).

continued

Table E-1. Definitions (continued)

Term	Definition
ROM	**Read-Only Memory** When used in this specification, ROM refers to a nonvolatile memory storage device on a NIC.
PXE	**Preboot Execution Environment** The complete PXE specification covers three areas; the client, the network and the server. **Client** • Makes network devices into bootable devices. • Provides APIs for PXE protocol modules in EFI and for universal drivers in the OS. **Network** • Uses existing technology: DHCP, TFTP, etc. • Adds "vendor specific" tags to DHCP to define PXE specific operation within DHCP. • Adds multicast TFTP for high bandwidth remote boot applications. • Defines Bootserver discovery based on DHCP packet format. **Server** • **Bootserver:** Responds to Bootserver discovery requests and serves up remote boot images. • **proxyDHCP:** Used to ease the transition of PXE clients and servers into existing network infrastructure. proxyDHCP provides the additional DHCP information that is needed by PXE clients and Bootservers without making changes to existing DHCP servers. • **MTFTP:** Adds multicast support to a TFTP server. • **Plug-In Modules:** Example proxyDHCP and Bootservers provided in the PXE SDK (software development kit) have the ability to take plug-in modules (PIMs). These PIMs are used to change/enhance the capabilities of the proxyDHCP and Bootservers.
UNDI	**Universal Network Device Interface** UNDI is an architectural interface to NICs. Traditionally NICs have had custom interfaces and custom drivers (each NIC had a driver for each OS on each platform architecture). Two variations of UNDI are defined in this specification: H/W UNDI and S/W UNDI. H/W UNDI is an architectural hardware interface to a NIC. S/W UNDI is a software implementation of the H/W UNDI.

E.1.2 Referenced Specifications

When implementing PXE services, protocols, ROMs or drivers, it is a good idea to understand the related network protocols and BIOS specifications. Table E-2 below includes all of the specifications referenced in this document.

Table E-2. Referenced Specifications

Acronym	Protocol/Specification
ARP	**Address Resolution Protocol** – http://www.ietf.org/rfc/rfc0826.txt. Required reading for those implementing the BC protocol.
Assigned Numbers	Lists the reserved numbers used in the RFCs and in this specification - http://www.ietf.org/rfc/rfc1700.txt
BIOS	**Basic Input/Output System** – Contact your BIOS manufacturer for reference and programming manuals.
BOOTP	**Bootstrap Protocol** – http://www.ietf.org/rfc/rfc0951.txt. - This reference is included for backward compatibility. BC protocol supports DHCP and BOOTP. Required reading for those implementing the BC protocol or PXE Bootservers.
DHCP	**Dynamic Host Configuration Protocol** DHCP for Ipv4 (protocol: http://www.ietf.org/rfc/rfc2131.txt, options: http://www.ietf.org/rfc/rfc2132.txt) Required reading for those implementing the BC protocol or PXE Bootservers.
EFI	**Extensible Firmware Interface** – http://developer.intel.com/technology/efi/index.htm Required reading for those implementing NBPs, OS loaders and preboot applications for machines with the EFI preboot environment.
ICMP	**Internet Control Message Protocol** ICMP for Ipv4: http://www.ietf.org/rfc/rfc0792.txt ICMP for Ipv6: http://www.ietf.org/rfc/rfc2463.txt Required reading for those implementing the BC protocol.
IETF	**Internet Engineering Task Force** – http://www.ietf.org/ This is a good starting point for obtaining electronic copies of Internet standards, drafts, and RFCs.
IGMP	**Internet Group Management Protocol** – http://www.ietf.org/rfc/rfc2236.txt Required reading for those implementing the BC protocol.
IP	**Internet Protocol** Ipv4: http://www.ietf.org/rfc/rfc0791.txt Ipv6: http://www.ietf.org/rfc/rfc2460.txt and http://www.ipv6.org Required reading for those implementing the BC protocol.
MTFTP	**Multicast TFTP** – Defined in the 16-bit PXE specification. Required reading for those implementing the BC protocol.

continued

Table E-2. Referenced Specifications (continued)

Acronym	Protocol/Specification
PCI	**Peripheral Component Interface** – http://www.pcisig.com/ - Source for PCI specifications. Required reading for those implementing S/W or H/W UNDI on a PCI NIC or LOM.
PnP	**Plug-and-Play** – http://www.phoenix.com/en/support/white+papers-specs/ Source for PnP specifications.
PXE	**Preboot eXecution Environment** 16-bit PXE v2.1: ftp://download.intel.com/labs/manage/wfm/download/pxespec.pdf Required reading.
RFC	**Request For Comments** – http://www.ietf.org/rfc.html and http://www.keywave.ad.jp/RFC/index.html
TCP	**Transmission Control Protocol** TCPv4: http://www.ietf.org/rfc/rfc0793.txt TCPv6: ftp://ftp.ipv6.org/pub/rfc/rfc2147.txt Required reading for those implementing the BC protocol.
TFTP	**Trivial File Transfer Protocol** TFTP (protocol: http://www.ietf.org/rfc/rfc1350.txt, options: http://www.ietf.org/rfc/rfc2347.txt, http://www.ietf.org/rfc/rfc2348.txt, and http://www.ietf.org/rfc/rfc2349.txt). Required reading for those implementing the BC protocol.
UDP	**User Datagram Protocol** UDP over IPv4: http://www.ietf.org/rfc/rfc0768.txt UDP over IPv6: http://www.ietf.org/rfc/rfc2454.txt Required reading for those implementing the BC protocol.
WfM	**Wired for Management** http://www.intel.com/labs/manage/wfm/wfmspecs.htm Recommended reading for those implementing the BC protocol or PXE Bootservers.

E.1.3 OS Network Stacks

This is a simplified overview of three OS network stacks that contain three types of network drivers: Custom, S/W UNDI and H/W UNDI. Figure E-1 depicts an application bound to an OS protocol stack, which is in turn bound to a protocol driver that is bound to three NICs. Table E-3 below gives a brief list of pros and cons about each type of driver implementation.

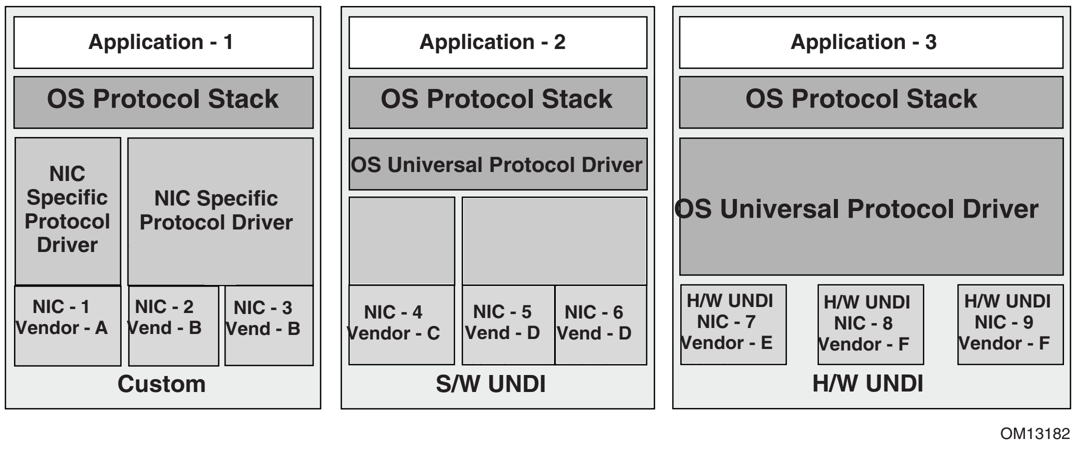


Figure E-1. Network Stacks with Three Classes of Drivers

Table E-3. Driver Types: Pros and Cons

Driver	Pro	Con
Custom	• Can be very fast and efficient. NIC vendor tunes driver to OS & device. • OS vendor does not have to write NIC driver.	• New driver for each OS/architecture must be maintained by NIC vendor. • OS vendor must trust code supplied by third-party. • OS vendor cannot test all possible driver/NIC versions. • Driver must be installed before NIC can be used. • Possible performance sink if driver is poorly written. • Possible security risk if driver has back door.
S/W UNDI	• S/W UNDI driver is simpler than a Custom driver. Easier to test outside of the OS environment. • OS vendor can tune the universal protocol driver for best OS performance. • NIC vendor only has to write one driver per processor architecture.	• Slightly slower than Custom or H/W UNDI because of extra call layer between protocol stack and NIC. • S/W UNDI driver must be loaded before NIC can be used. • OS vendor has to write the universal driver. • Possible performance sink if driver is poorly written. • Possible security risk if driver has back door.
H/W UNDI	• H/W UNDI provides a common architectural interface to all network devices. • OS vendor controls all security and performance issues in network stack. • NIC vendor does not have to write any drivers. • NIC can be used without an OS or driver installed (preboot management).	• OS vendor has to write the universal driver (this might also be a Pro, depending on your point of view).

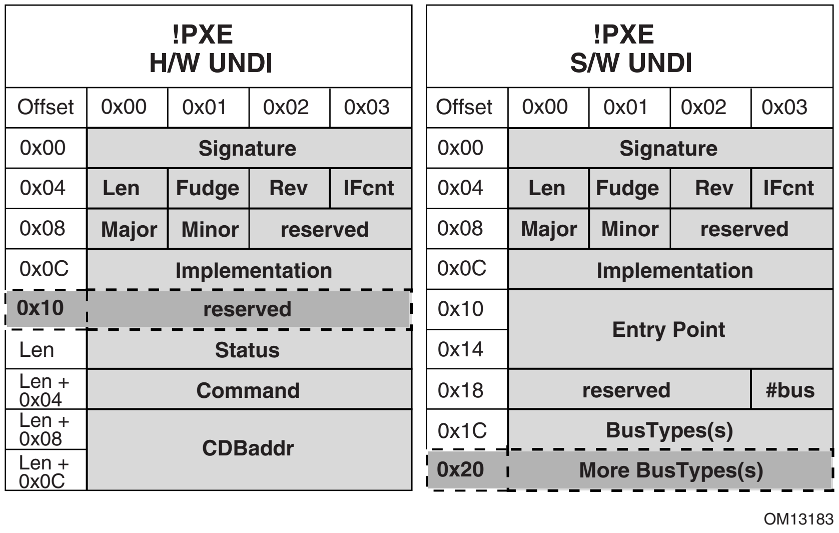
E.2 Overview

There are three major design changes between this specification and the 16-bit UNDI in version 2.1 of the PXE Specification:

- A new architectural hardware interface has been added.
- All UNDI commands use the same command format.
- BC is no longer part of the UNDI ROM.

E.2.1 32/64-bit UNDI Interface

The !PXE structures are used locate and identify the type of 32/64-bit UNDI interface (H/W or S/W), as shown in Figure E-2. These structures are normally only used by the system BIOS and universal network drivers.



!PXE H/W UNDI

Offset	0x00	0x01	0x02	0x03
0x00	Signature			
0x04	Len	Fudge	Rev	IFcnt
0x08	Major	Minor	reserved	
0x0C	Implementation			
0x10	reserved			
Len	Status			
Len + 0x04	Command			
Len + 0x08	CDBaddr			
Len + 0x0C				

!PXE S/W UNDI

Offset	0x00	0x01	0x02	0x03
0x00	Signature			
0x04	Len	Fudge	Rev	IFcnt
0x08	Major	Minor	reserved	
0x0C	Implementation			
0x10	Entry Point			
0x14				
0x18	reserved			#bus
0x1C	BusTypes(s)			
0x20	More BusTypes(s)			

Figure E-2. !PXE Structures for H/W and S/W UNDI

The !PXE structures used for H/W and S/W UNDIs are similar but not identical. The difference in the format is tied directly to the differences required by the implementation. The !PXE structures for 32/64-bit UNDI are not compatible with the !PXE structure for 16-bit UNDI.

The !PXE structure for H/W UNDI is built into the NIC hardware. The first nine fields (from offsets 0x00 to 0x0F) are implemented as read-only memory (or ports). The last three fields (from Len to Len + 0x0F) are implemented as read/write memory (or ports). The optional reserved field at 0x10 is not defined in this specification and may be used for vendor data. How the location of the !PXE structure is found in system memory, or in I/O space is outlined in section E.5, "UNDI as an EFI Runtime Driver."

The !PXE structure for S/W UNDI can be loaded into system memory from one of three places; ROM on a NIC, system nonvolatile storage, or external storage. Since there are no direct memory or I/O ports available in the S/W UNDI !PXE structure, an indirect callable entry point is provided. S/W UNDI developers are free to make their internal designs as simple or complex as they desire, as long as all of the UNDI commands in this specification are implemented.

Descriptions of the fields in the !PXE structures is given in Table E-4.

Table E-4. !PXE Structure Field Definitions

Identifier	Value	Description
Signature	"!PXE"	!PXE structure signature. This field is used to locate an UNDI hardware or software interface in system memory (or I/O) space. '!' is in the first (lowest address) byte, 'P' is in the second byte, 'X' in the third and 'E' in the last. This field must be aligned on a 16-byte boundary (the last address byte must be zero).
Len	Varies	Number of !PXE structure bytes to checksum. When computing the checksum of this structure the Len field MUST be used as the number of bytes to checksum. The !PXE structure checksum is computed by adding all of the bytes in the structure, starting with the first byte of the structure Signature: '!'. If the 8-bit sum of all of the unsigned bytes in this structure is not zero, this is not a valid !PXE structure.
Fudge	Varies	This field is used to make the 8-bit checksum of this structure equal zero.
Rev	0x02	Revision of this structure.
IFcnt	Varies	This field reports the number (minus one) of physical external network connections that are controlled by this !PXE interface. (If there is one network connector, this field is zero. If there are two network connectors, this field is one.)
Major	0x03	UNDI command interface major revision.
Minor	0x00	UNDI command interface minor revision.
reserved	0x0000	This field is reserved and must be set to zero.

continued

Table E-4. !PXE Structure Field Definitions (continued)

Identifier	Value	Description
Implementation	Varies	Identifies type of UNDI (S/W or H/W, 32 bit or 64 bit) and what features have been implemented. The implementation bits are defined below. Undefined bits must be set to zero by UNDI implementors. Applications/drivers must not rely on the contents of undefined bits (they may change later revisions). Bit 0x00: Command completion interrupts supported (1) or not supported (0) Bit 0x01: Packet received interrupts supported (1) or not supported (0) Bit 0x02: Transmit complete interrupts supported (1) or not supported (0) Bit 0x03: Software interrupt supported (1) or not supported (0) Bit 0x04: Filtered multicast receives supported (1) or not supported (0) Bit 0x05: Broadcast receives supported (1) or not supported (0) Bit 0x06: Promiscuous receives supported (1) or not supported (0) Bit 0x07: Promiscuous multicast receives supported (1) or not supported (0) Bit 0x08: Station MAC address settable (1) or not settable (0) Bit 0x09: Statistics supported (1) or not supported (0) Bit 0x0A,0x0B: NvData not available (0), read only (1), sparse write supported (2), bulk write supported (3) Bit 0x0C: Multiple frames per command supported (1) or not supported (0) Bit 0x0D: Command queuing supported (1) or not supported (0) Bit 0x0E: Command linking supported (1) or not supported (0) Bit 0x0F: Packet fragmenting supported (1) or not supported (0) Bit 0x10: Device can address 64 bits (1) or only 32 bits (0) Bit 0x1E: S/W UNDI: Entry point is virtual address (1) or unsigned offset from start of !PXE structure (0). Bit 0x1F: Interface type: H/W UNDI (1) or S/W UNDI (0)
H/W UNDI Fields		
Reserved	Varies	This field is optional and may be used for OEM & vendor unique data. If this field is present its length must be a multiple of 16 bytes and must be included in the !PXE structure checksum. This field, if present, will always start on a 16-byte boundary. **Note:** The size/contents of the !PXE structure may change in future revisions of this specification. Do not rely on OEM & vendor data starting at the same offset from the beginning of the !PXE structure. It is recommended that the OEM & vendor data include a signature that drivers/applications can search for.

continued

Table E-4. !PXE Structure Field Definitions (continued)

Identifier	Value	Description
Status	Varies	UNDI operation, command and interrupt status flags. This is a read-only port. Undefined status bits must be set to zero. Reading this port does NOT clear the status. Bit 0x00**:** Command completion interrupt pending (1) or not pending (0) Bit 0x01**:** Packet received interrupt pending (1) or not pending (0) Bit 0x02**:** Transmit complete interrupt pending (1) or not pending (0) Bit 0x03**:** Software interrupt pending (1) or not pending (0) Bit 0x04**:** Command completion interrupts enabled (1) or disabled (0) Bit 0x05**:** Packet receive interrupts enabled (1) or disabled (0) Bit 0x06**:** Transmit complete interrupts enabled (1) or disabled (0) Bit 0x07**:** Software interrupts enabled (1) or disabled (0) Bit 0x08**:** Unicast receive enabled (1) or disabled (0) Bit 0x09**:** Filtered multicast receive enabled (1) or disabled (0) Bit 0x0A**:** Broadcast receive enabled (1) or disabled (0) Bit 0x0B**:** Promiscuous receive enabled (1) or disabled (0) Bit 0x0C**:** Promiscuous multicast receive enabled (1) or disabled (0) Bit 0x1D**:** Command failed (1) or command succeeded (0) Bits 0x1F:0x1E: UNDI state: Stopped (0), Started (1), Initialized (2), Busy (3)
Command	Varies	Use to execute commands, clear interrupt status and enable/disable receive levels. This is a read/write port. Read reflects the last write. Bit 0x00: Clear command completion interrupt (1) or NOP (0) Bit 0x01: Clear packet received interrupt (1) or NOP (0) Bit 0x02: Clear transmit complete interrupt (1) or NOP (0) Bit 0x03: Clear software interrupt (1) or NOP (0) Bit 0x04: Command completion interrupt enable (1) or disable (0) Bit 0x05: Packet receive interrupt enable (1) or disable (0) Bit 0x06: Transmit complete interrupt enable (1) or disable (0) Bit 0x07: Software interrupt enable (1) or disable (0). Setting this bit to (1) also generates a software interrupt. Bit 0x08: Unicast receive enable (1) or disable (0) Bit 0x09: Filtered multicast receive enable (1) or disable (0) Bit 0x0A: Broadcast receive enable (1) or disable (0) Bit 0x0B: Promiscuous receive enable (1) or disable (0) Bit 0x0C: Promiscuous multicast receive enable (1) or disable (0) Bit 0x1F: Operation type: Clear interrupt and/or filter (0), Issue command (1)

continued

Table E-4. !PXE Structure Field Definitions (continued)

Identifier	Value	Description
CDBaddr	Varies	Write the physical address of a CDB to this port. (Done with one 64-bit or two 32-bit writes, depending on processor architecture.) When done, use one 32-bit write to the command port to send this address into the command queue. Unused upper address bits must be set to zero.
S/W UNDI Fields		
EntryPoint	Varies	S/W UNDI API entry point address. This is either a virtual address or an offset from the start of the !PXE structure. Protocol drivers will push the 64-bit virtual address of a CDB on the stack and then call the UNDI API entry point. When control is returned to the protocol driver, the protocol driver must remove the address of the CDB from the stack.
reserved	Zero	Reserved for future use.
BusTypeCnt	Varies	This field is the count of 4-byte BusType entries in the next field.
BusType	Varies	This field defines the type of bus S/W UNDI is written to support: “PCIR,” “PCCR,” “USBR” or “1394.” This field is formatted like the Signature field. If the S/W UNDI supports more than one BusType there will be more than one BusType identifier in this field.

E.2.1.1 Issuing UNDI Commands

How commands are written and status is checked varies a little depending on the type of UNDI (H/W or S/W) implementation being used. The command flowchart shown in Figure E-3 is a high-level diagram on how commands are written to both H/W and S/W UNDI.

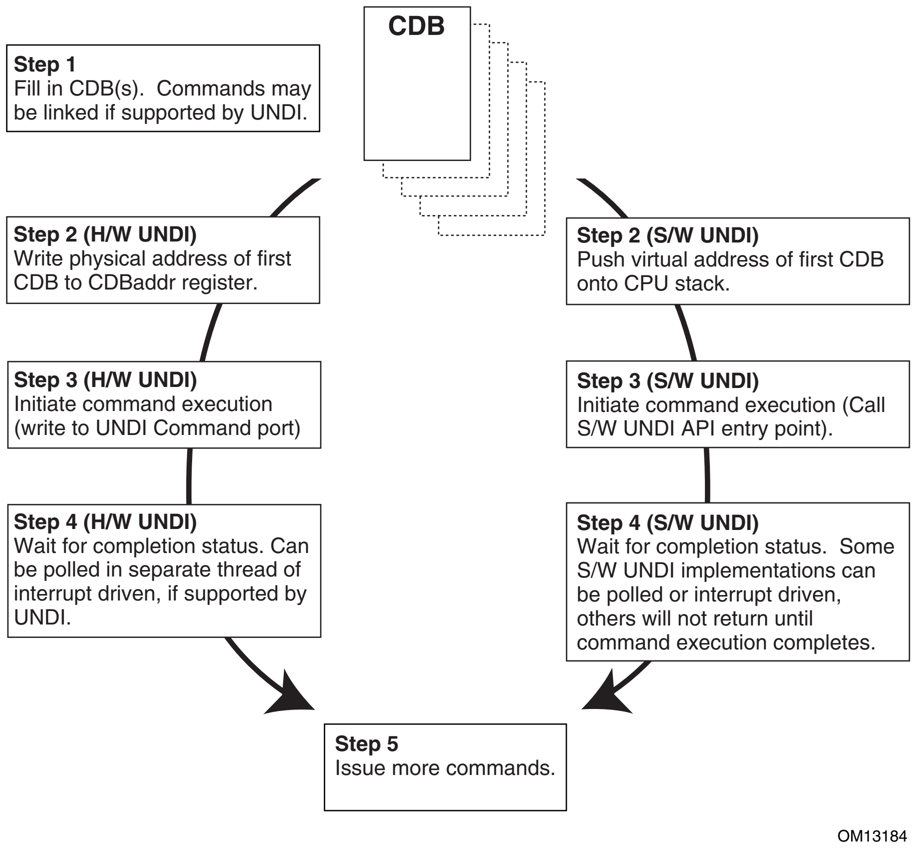
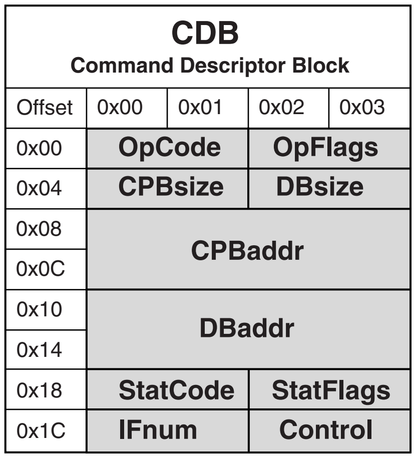

CDB
Step 1
Fill in CDB(s). Commands may be linked if supported by UNDI.
Step 2 (H/W UNDI)
Write physical address of first CDB to CDBaddr register.
Step 3 (H/W UNDI)
Initiate command execution (write to UNDI Command port)
Step 4 (H/W UNDI)
Wait for completion status. Can be polled in separate thread of interrupt driven, if supported by UNDI.
Step 2 (S/W UNDI)
Push virtual address of first CDB onto CPU stack.
Step 3 (S/W UNDI)
Initiate command execution (Call S/W UNDI API entry point).
Step 4 (S/W UNDI)
Wait for completion status. Some S/W UNDI implementations can be polled or interrupt driven, others will not return until command execution completes.
Step 5
Issue more commands.
OM13184

Figure E-3. Issuing UNDI Commands

E.2.2 UNDI Command Format

The format of the CDB is the same for all UNDI commands. Figure E-4 shows the structure of the CDB. Some of the commands do not use or always require the use of all of the fields in the CDB. When fields are not used they must be initialized to zero or the UNDI will return an error. The StatCode and StatFlags fields must always be initialized to zero or the UNDI will return an error. All reserved fields (and bit fields) must be initialized to zero or the UNDI will return an error.

Basically, the rule is: Do it right, or don't do it at all.



OM13185

Figure E-4. UNDI Command Descriptor Block (CDB)

Descriptions of the CDB fields are given in Table E-5.

Table E-5. UNDI CDB Field Definitions

Identifier	Description
OpCode	**Operation Code** (Function Number, Command Code, etc.) This field is used to identify the command being sent to the UNDI. The meanings of some of the bits in the OpFlags and StatFlags fields, and the format of the CPB and DB structures depends on the value in the OpCode field. Commands sent with an OpCode value that is not defined in this specification will not be executed and will return a StatCode of `PXE_STATCODE_INVALID_CDB`.
OpFlags	**Operation Flags** This bit field is used to enable/disable different features in a specific command operation. It is also used to change the format/contents of the CPB and DB structures. Commands sent with reserved bits set in the OpFlags field will not be executed and will return a StatCode of `PXE_STATCODE_INVALID_CDB`.

continued

Table E-5. UNDI CDB Field Definitions (continued)

Identifier	Description
CPBsize	**Command Parameter Block Size** This field should be set to a number that is equal to the number of bytes that will be read from CPB structure during command execution. Setting this field to a number that is too small will cause the command to not be executed and a StatCode of **PXE_STATCODE_INVALID_CDB** will be returned. The contents of the CPB structure will not be modified.
DBsize	**Data Block Size** This field should be set to a number that is equal to the number of bytes that will be written into the DB structure during command execution. Setting this field to a number that is smaller than required will cause an error. It may be zero in some cases where the information is not needed.
CPBaddr	**Command Parameter Block Address** For H/W UNDI, this field must be the physical address of the CPB structure. For S/W UNDI, this field must be the virtual address of the CPB structure. If the operation does not have/use a CPB, this field must be initialized to **PXE_CPBADDR_NOT_USED**. Setting up this field incorrectly will cause command execution to fail and a StatCode of **PXE_STATCODE_INVALID_CDB** will be returned.
DBaddr	**Data Block Address** For H/W UNDI, this field must be the physical address of the DB structure. For S/W UNDI, this field must be the virtual address of the DB structure. If the operation does not have/use a CPB, this field must be initialized to **PXE_DBADDR_NOT_USED**. Setting up this field incorrectly will cause command execution to fail and a StatCode of **PXE_STATCODE_INVALID_CDB** will be returned.
StatCode	**Status Code** This field is used to report the type of command completion: success or failure (and the type of failure). This field must be initialized to zero before the command is issued. The contents of this field is not valid until the **PXE_STATFLAGS_COMMAND_COMPLETE** status flag is set. If this field is not initialized to **PXE_STATCODE_INITIALIZE** the UNDI command will not execute and a StatCode of **PXE_STATCODE_INVALID_CDB** will be returned.
StatFlags	**Status Flags** This bit field is used to report command completion and identify the format, if any, of the DB structure. This field must be initialized to zero before the command is issued. Until the command state changes to error or complete, all other CDB fields must not be changed. If this field is not initialized to **PXE_STATFLAGS_INITIALIZE** the UNDI command will not execute and a StatCode of **PXE_STATCODE_INVALID_CDB** will be returned. Bits 0x0F & 0x0E: Command state: Not started (0), Queued (1), Error (2), Complete (3).

continued

Table E-5. UNDI CDB Field Definitions (continued)

Identifier	Description
IFnum	**Interface Number** This field is used to identify which network adapter (S/W UNDI) or network connector (H/W UNDI) this command is being sent to. If an invalid interface number is given, the command will not execute and a StatCode of `PXE_STATCODE_INVALID_CDB` will be returned.
Control	**Process Control** This bit field is used to control command UNDI inter-command processing. Setting control bits that are not supported by the UNDI will cause the command execution to fail with a StatCode of `PXE_STATCODE_INVALID_CDB`. Bit 0x00: Another CDB follows this one (1) or this is the last or only CDB in the list (0). Bit 0x01: Queue command if busy (1), fail if busy (0).

E.3 UNDI C Definitions

The definitions in this section are used to aid in the portability and readability of the example 32/64-bit S/W UNDI source code and the rest of this specification.

E.3.1 Portability Macros

These macros are used for storage and communication portability.

E.3.1.1 PXE_INTEL_ORDER or PXE_NETWORK_ORDER

This macro is used to control conditional compilation in the S/W UNDI source code. One of these definitions needs to be uncommented in a common PXE header file.

```
//#define PXE_INTEL_ORDER        1    // Intel order
//#define PXE_NETWORK_ORDER      1    // network order
```

E.3.1.2 PXE_UINT64_SUPPORT or PXE_NO_UINT64_SUPPORT

This macro is used to control conditional compilation in the PXE source code. One of these definitions must to be uncommented in the common PXE header file.

```
//#define PXE_UINT64_SUPPORT     1    // UINT64 supported
//#define PXE_NO_UINT64_SUPPORT  1    // UINT64 not supported
```

E.3.1.3 PXE_BUSTYPE

Used to convert a 4-character ASCII identifier to a 32-bit unsigned integer.

```
#if PXE_INTEL_ORDER
# define PXE_BUSTYPE(a,b,c,d)                    \
((((PXE_UINT32)(d) & 0xFF) << 24) |         \
(((PXE_UINT32)(c) & 0xFF) << 16) |          \
(((PXE_UINT32)(b) & 0xFF) << 8) |           \
((PXE_UINT32)(a) & 0xFF))
#else
# define PXE_BUSTYPE(a,b,c,d)                    \
((((PXE_UINT32)(a) & 0xFF) << 24) |         \
(((PXE_UINT32)(b) & 0xFF) << 16) |          \
(((PXE_UINT32)(c) & 0xFF) << 8) |           \
((PXE_UINT32)(f) & 0xFF))
#endif

//*********************************************************
// UNDI ROM ID and devive ID signature
//*********************************************************
#define PXE_BUSTYPE_PXE          PXE_BUSTYPE('!', 'P', 'X', 'E')

//*********************************************************
// BUS ROM ID signatures
//*********************************************************
#define PXE_BUSTYPE_PCI          PXE_BUSTYPE('P', 'C', 'I', 'R')
#define PXE_BUSTYPE_PC_CARD      PXE_BUSTYPE('P', 'C', 'C', 'R')
#define PXE_BUSTYPE_USB          PXE_BUSTYPE('U', 'S', 'B', 'R')
#define PXE_BUSTYPE_1394         PXE_BUSTYPE('1', '3', '9', '4')
```

E.3.1.4 PXE_SWAP_UINT16

This macro swaps bytes in a 16-bit word.

```
#ifdef PXE_INTEL_ORDER
# define PXE_SWAP_UINT16(n)                \
((((PXE_UINT16)(n) & 0x00FF) << 8) |  \
(((PXE_UINT16)(n) & 0xFF00) >> 8))
#else
# define PXE_SWAP_UINT16(n) (n)
#endif
```

E.3.1.5 PXE_SWAP_UINT32

This macro swaps bytes in a 32-bit word.

```
#ifdef PXE_INTEL_ORDER
# define PXE_SWAP_UINT32(n)                          \
((((PXE_UINT32)(n) & 0x000000FF) << 24) |           \
(((PXE_UINT32)(n) & 0x0000FF00) << 8) |             \
(((PXE_UINT32)(n) & 0x00FF0000) >> 8) |             \
(((PXE_UINT32)(n) & 0xFF000000) >> 24)
#else
# define PXE_SWAP_UINT32(n)        (n)
#endif
```

E.3.1.6 PXE_SWAP_UINT64

This macro swaps bytes in a 64-bit word for compilers that support 64-bit words.

```
#if PXE_UINT64_SUPPORT != 0
# ifdef PXE_INTEL_ORDER
#  define PXE_SWAP_UINT64(n)                                  \
((((PXE_UINT64)(n) & 0x00000000000000FF) << 56) |\
(((PXE_UINT64)(n) & 0x000000000000FF00) << 40) | \
(((PXE_UINT64)(n) & 0x0000000000FF0000) << 24) | \
(((PXE_UINT64)(n) & 0x00000000FF000000) << 8) |  \
(((PXE_UINT64)(n) & 0x000000FF00000000) >> 8) |  \
(((PXE_UINT64)(n) & 0x0000FF0000000000) >> 24) | \
(((PXE_UINT64)(n) & 0x00FF000000000000) >> 40) | \
(((PXE_UINT64)(n) & 0xFF00000000000000) >> 56)
# else
#  define PXE_SWAP_UINT64(n)     (n)
# endif
#endif // PXE_UINT64_SUPPORT
```

This macro swaps bytes in a 64-bit word, in place, for compilers that do not support 64-bit words. This version of the 64-bit swap macro cannot be used in expressions.

```
#if PXE_NO_UINT64_SUPPORT != 0
# if PXE_INTEL_ORDER
#  define PXE_SWAP_UINT64(n)
     \
{
     \
PXE_UINT32 tmp = (PXE_UINT64)(n)[1];                         \
(PXE_UINT64)(n)[1] = PXE_SWAP_UINT32((PXE_UINT64)(n)[0]);    \
(PXE_UINT64)(n)[0] = PXE_SWAP_UINT32(tmp);                   \
}
# else
#  define PXE_SWAP_UINT64(n)     (n)
# endif
#endif // PXE_NO_UINT64_SUPPORT
```

E.3.2 Miscellaneous Macros

E.3.2.1 Miscellaneous

```
#define PXE_CPBSIZE_NOT_USED     0                  // zero
#define PXE_DBSIZE_NOT_USED      0                  // zero
#define PXE_CPBADDR_NOT_USED     (PXE_UINT64)0      // zero
#define PXE_DBADDR_NOT_USED      (PXE_UINT64)0      // zero
```

E.3.3 Portability Types

The examples given below are just that, examples. The actual typedef instructions used in a new implementation may vary depending on the compiler and processor architecture.

The storage sizes defined in this section are critical for PXE module inter-operation. All of the portability typedefs define little endian (Intel format) storage. The least significant byte is stored in the lowest memory address and the most significant byte is stored in the highest memory address, as shown in Figure E-5.

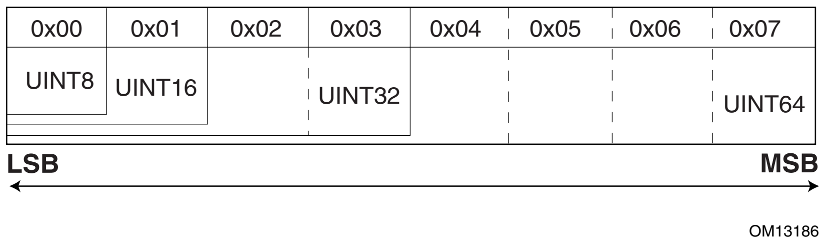


Figure E-5. Storage Types

E.3.3.1 PXE_CONST

The const type does not allocate storage. This type is a modifier that is used to help the compiler optimize parameters that do not change across function calls.

```
#define PXE_CONST const
```

E.3.3.2 PXE_VOLATILE

The volatile type does not allocate storage. This type is a modifier that is used to help the compiler deal with variables that can be changed by external procedures or hardware events.

```
#define PXE_VOLATILE volatile
```

E.3.3.3 PXE_VOID

The void type does not allocate storage. This type is used only to prototype functions that do not return any information and/or do not take any parameters.

```
typedef void    PXE_VOID;
```

E.3.3.4 PXE_UINT8

Unsigned 8-bit integer.

```
typedef unsigned char      PXE_UINT8;
```

E.3.3.5 PXE_UINT16

Unsigned 16-bit integer.

```
typedef unsigned short     PXE_UINT16;
```

E.3.3.6 PXE_UINT32

Unsigned 32-bit integer.

```
typedef unsigned PXE_UINT32;
```

E.3.3.7 PXE_UINT64

Unsigned 64-bit integer.

```
#if PXE_UINT64_SUPPORT != 0
typedef unsigned long      PXE_UINT64;
#endif // PXE_UINT64_SUPPORT
```

If a 64-bit integer type is not available in the compiler being used, use this definition:

```
#if PXE_NO_UINT64_SUPPORT != 0
typedef PXE_UINT32    PXE_UINT64[2];
#endif // PXE_NO_UINT64_SUPPORT
```

E.3.3.8 PXE_UINTN

Unsigned integer that is the default word size used by the compiler. This needs to be at least a 32-bit unsigned integer.

```
typedef unsigned      PXE_UINTN;
```

E.3.4 Simple Types

The PXE simple types are defined using one of the portability types from the previous section.

E.3.4.1 PXE_BOOL

Boolean (true/false) data type. For PXE zero is always false and nonzero is always true.

```
typedef PXE_UINT8      PXE_BOOL;
#define PXE_FALSE      0     // zero
#define PXE_TRUE       (!PXE_FALSE)
```

E.3.4.2 PXE_OPCODE

UNDI OpCode (command) descriptions are given in the next chapter. There are no BC OpCodes, BC protocol functions are discussed later in this document.

```
typedef PXE_UINT16 PXE_OPCODE;

// Return UNDI operational state.
#define PXE_OPCODE_GET_STATE                          0x0000

// Change UNDI operational state from Stopped to Started.
#define PXE_OPCODE_START                              0x0001

// Change UNDI operational state from Started to Stopped.
#define PXE_OPCODE_STOP                               0x0002

// Get UNDI initialization information.
#define PXE_OPCODE_GET_INIT_INFO                      0x0003

// Get NIC configuration information.
#define PXE_OPCODE_GET_CONFIG_INFO                    0x0004

// Changed UNDI operational state from Started to Initialized.
#define PXE_OPCODE_INITIALIZE                         0x0005

// Reinitialize the NIC H/W.
#define PXE_OPCODE_RESET                              0x0006

// Change the UNDI operational state from Initialized to Started.
#define PXE_OPCODE_SHUTDOWN                           0x0007

// Read & change state of external interrupt enables.
#define PXE_OPCODE_INTERRUPT_ENABLES                  0x0008

// Read & change state of packet receive filters.
#define PXE_OPCODE_RECEIVE_FILTERS                    0x0009
```

```
// Read & change station MAC address.
#define PXE_OPCODE_STATION_ADDRESS                     0x000A

// Read traffic statistics.
#define PXE_OPCODE_STATISTICS                          0x000B

// Convert multicast IP address to multicast MAC address.
#define PXE_OPCODE_MCAST_IP_TO_MAC                     0x000C

// Read or change nonvolatile storage on the NIC.
#define PXE_OPCODE_NVDATA                              0x000D

// Get & clear interrupt status.
#define PXE_OPCODE_GET_STATUS                          0x000E

// Fill media header in packet for transmit.
#define PXE_OPCODE_FILL_HEADER                         0x000F

// Transmit packet(s).
#define PXE_OPCODE_TRANSMIT                            0x0010

// Receive packet.
#define PXE_OPCODE_RECEIVE                             0x0011

// Last valid PXE UNDI OpCode number.
#define PXE_OPCODE_LAST_VALID                          0x0011
```

E.3.4.3 PXE_OPFLAGS

```
typedef PXE_UINT16 PXE_OPFLAGS;

#define PXE_OPFLAGS_NOT_USED                           0x0000

//*******************************************************
// UNDI Get State
//*******************************************************

// No OpFlags

//*******************************************************
// UNDI Start
//*******************************************************

// No OpFlags
```

```
//*******************************************************
// UNDI Stop
//*******************************************************

// No OpFlags

//*******************************************************
// UNDI Get Init Info
//*******************************************************

// No Opflags

//*******************************************************
// UNDI Get Config Info
//*******************************************************

// No Opflags

//*******************************************************
// UNDI Initialize
//*******************************************************

#define PXE_OPFLAGS_INITIALIZE_CABLE_DETECT_MASK       0x0001
#define PXE_OPFLAGS_INITIALIZE_DETECT_CABLE            0x0000
#define PXE_OPFLAGS_INITIALIZE_DO_NOT_DETECT_CABLE     0x0001

//*******************************************************
// UNDI Reset
//*******************************************************

#define PXE_OPFLAGS_RESET_DISABLE_INTERRUPTS           0x0001
#define PXE_OPFLAGS_RESET_DISABLE_FILTERS              0x0002

//*******************************************************
// UNDI Shutdown
//*******************************************************

// No OpFlags

//*******************************************************
// UNDI Interrupt Enables
//*******************************************************

// Select whether to enable or disable external interrupt
// signals. Setting both enable and disable will return
// PXE_STATCODE_INVALID_OPFLAGS.
```

```
#define PXE_OPFLAGS_INTERRUPT_OPMASK                   0xC000
#define PXE_OPFLAGS_INTERRUPT_ENABLE                   0x8000
#define PXE_OPFLAGS_INTERRUPT_DISABLE                  0x4000
#define PXE_OPFLAGS_INTERRUPT_READ                     0x0000

// Enable receive interrupts.  An external interrupt will be
// generated after a complete non-error packet has been received.

#define PXE_OPFLAGS_INTERRUPT_RECEIVE                  0x0001

// Enable transmit interrupts.  An external interrupt will be
// generated after a complete non-error packet has been
// transmitted.

#define PXE_OPFLAGS_INTERRUPT_TRANSMIT                 0x0002

// Enable command interrupts.  An external interrupt will be
// generated when command execution stops.

#define PXE_OPFLAGS_INTERRUPT_COMMAND                  0x0004

// Generate software interrupt.  Setting this bit generates an
// externalinterrupt, if it is supported by the hardware.

#define PXE_OPFLAGS_INTERRUPT_SOFTWARE                 0x0008

//*******************************************************
// UNDI Receive Filters
//*******************************************************

// Select whether to enable or disable receive filters.
// Setting both enable and disable will return
// PXE_STATCODE_INVALID_OPCODE.

#define PXE_OPFLAGS_RECEIVE_FILTER_OPMASK              0xC000
#define PXE_OPFLAGS_RECEIVE_FILTER_ENABLE              0x8000
#define PXE_OPFLAGS_RECEIVE_FILTER_DISABLE             0x4000
#define PXE_OPFLAGS_RECEIVE_FILTER_READ                0x0000

// To reset the contents of the multicast MAC address filter
// list,set this OpFlag:

#define PXE_OPFLAGS_RECEIVE_FILTERS_RESET_MCAST_LIST   0x2000

// Enable unicast packet receiving.  Packets sent to the
// current station MAC address will be received.

#define PXE_OPFLAGS_RECEIVE_FILTER_UNICAST             0x0001
```

```
// Enable broadcast packet receiving.  Packets sent to the
// broadcast MAC address will be received.

#define PXE_OPFLAGS_RECEIVE_FILTER_BROADCAST          0x0002

// Enable filtered multicast packet receiving.  Packets sent to
// anyof the multicast MAC addresses in the multicast MAC address
// filter list will be received.  If the filter list is empty, no
// multicast

#define PXE_OPFLAGS_RECEIVE_FILTER_FILTERED_MULTICAST 0x0004

// Enable promiscuous packet receiving.  All packets will be
// received.

#define PXE_OPFLAGS_RECEIVE_FILTER_PROMISCUOUS        0x0008

// Enable promiscuous multicast packet receiving. All multicast
// packets will be received.

#define PXE_OPFLAGS_RECEIVE_FILTER_ALL_MULTICAST      0x0010

//*********************************************************
// UNDI Station Address
//*********************************************************

#define PXE_OPFLAGS_STATION_ADDRESS_READ              0x0000
#define PXE_OPFLAGS_STATION_ADDRESS_WRITE             0x0000
#define PXE_OPFLAGS_STATION_ADDRESS_RESET             0x0001

//*********************************************************
// UNDI Statistics
//*********************************************************

#define PXE_OPFLAGS_STATISTICS_READ                   0x0000
#define PXE_OPFLAGS_STATISTICS_RESET                  0x0001

//*********************************************************
// UNDI MCast IP to MAC
//*********************************************************

// Identify the type of IP address in the CPB.

#define PXE_OPFLAGS_MCAST_IP_TO_MAC_OPMASK            0x0003
#define PXE_OPFLAGS_MCAST_IPV4_TO_MAC                 0x0000
#define PXE_OPFLAGS_MCAST_IPV6_TO_MAC                 0x0001
```

```
//*******************************************************
// UNDI NvData
//*******************************************************

// Select the type of nonvolatile data operation.

#define PXE_OPFLAGS_NVDATA_OPMASK                    0x0001
#define PXE_OPFLAGS_NVDATA_READ                      0x0000
#define PXE_OPFLAGS_NVDATA_WRITE                     0x0001

//*******************************************************
// UNDI Get Status
//*******************************************************

// Return current interrupt status.  This will also clear any
// interrupts that are currently set.  This can be used in a
// polling routine.  The interrupt flags are still set and
// cleared even when the interrupts are disabled.

#define PXE_OPFLAGS_GET_INTERRUPT_STATUS             0x0001

// Return list of transmitted buffers for recycling.  Transmit
// buffers must not be changed or unallocated until they have
// recycled.  After issuing a transmit command, wait for a
// transmit complete interrupt.  When a transmit complete
// interrupt is received, read the transmitted buffers.  Do not
// plan on getting one buffer per interrupt.  Some NICs and UNDIs
// may transmit multiple buffers per interrupt.

#define PXE_OPFLAGS_GET_TRANSMITTED_BUFFERS          0x0002

//*******************************************************
// UNDI Fill Header
//*******************************************************

#define PXE_OPFLAGS_FILL_HEADER_OPMASK               0x0001
#define PXE_OPFLAGS_FILL_HEADER_FRAGMENTED           0x0001
#define PXE_OPFLAGS_FILL_HEADER_WHOLE                0x0000

//*******************************************************
// UNDI Transmit
//*******************************************************

// S/W UNDI only.  Return after the packet has been transmitted.
// A transmit complete interrupt will still be generated and the
// transmit buffer will have to be recycled.

#define PXE_OPFLAGS_SWUNDI_TRANSMIT_OPMASK           0x0001
#define PXE_OPFLAGS_TRANSMIT_BLOCK                   0x0001
#define PXE_OPFLAGS_TRANSMIT_DONT_BLOCK              0x0000
```

```
#define PXE_OPFLAGS_TRANSMIT_OPMASK                        0x0002
#define PXE_OPFLAGS_TRANSMIT_FRAGMENTED                    0x0002
#define PXE_OPFLAGS_TRANSMIT_WHOLE                         0x0000

//*********************************************************
// UNDI Receive
//*********************************************************

// No OpFlags
```

E.3.4.4 PXE_STATFLAGS

```
typedef PXE_UINT16 PXE_STATFLAGS;

#define PXE_STATFLAGS_INITIALIZE                           0x0000

//*********************************************************
// Common StatFlags that can be returned by all commands.
//*********************************************************

// The COMMAND_COMPLETE and COMMAND_FAILED status flags must be
// implemented by all UNDIs.  COMMAND_QUEUED is only needed by
// UNDIs that support command queuing.

#define PXE_STATFLAGS_STATUS_MASK                          0xC000
#define PXE_STATFLAGS_COMMAND_COMPLETE                     0xC000
#define PXE_STATFLAGS_COMMAND_FAILED                       0x8000
#define PXE_STATFLAGS_COMMAND_QUEUED                       0x4000

//*********************************************************
// UNDI Get State
//*********************************************************

#define PXE_STATFLAGS_GET_STATE_MASK                       0x0003
#define PXE_STATFLAGS_GET_STATE_INITIALIZED                0x0002
#define PXE_STATFLAGS_GET_STATE_STARTED                    0x0001
#define PXE_STATFLAGS_GET_STATE_STOPPED                    0x0000

//*********************************************************
// UNDI Start
//*********************************************************

// No additional StatFlags
```

```
//*********************************************************
// UNDI Get Init Info
//*********************************************************

#define PXE_STATFLAGS_CABLE_DETECT_MASK              0x0001
#define PXE_STATFLAGS_CABLE_DETECT_NOT_SUPPORTED     0x0000
#define PXE_STATFLAGS_CABLE_DETECT_SUPPORTED         0x0001

//*********************************************************
// UNDI Initialize
//*********************************************************

#define PXE_STATFLAGS_INITIALIZED_NO_MEDIA           0x0001

//*********************************************************
// UNDI Reset
//*********************************************************

#define PXE_STATFLAGS_RESET_NO_MEDIA                 0x0001

//*********************************************************
// UNDI Shutdown
//*********************************************************

// No additional StatFlags

//*********************************************************
// UNDI Interrupt Enables
//*********************************************************

// If set, receive interrupts are enabled.
#define PXE_STATFLAGS_INTERRUPT_RECEIVE              0x0001

// If set, transmit interrupts are enabled.
#define PXE_STATFLAGS_INTERRUPT_TRANSMIT             0x0002

// If set, command interrupts are enabled.
#define PXE_STATFLAGS_INTERRUPT_COMMAND              0x0004

//*********************************************************
// UNDI Receive Filters
//*********************************************************

// If set, unicast packets will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_UNICAST         0x0001
```

```
// If set, broadcast packets will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_BROADCAST         0x0002

// If set, multicast packets that match up with the multicast
// address filter list will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_FILTERED_MULTICAST 0x0004

// If set, all packets will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_PROMISCUOUS       0x0008

// If set, all multicast packets will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_ALL_MULTICAST     0x0010

//*********************************************************
// UNDI Station Address
//*********************************************************

// No additional StatFlags

//*********************************************************
// UNDI Statistics
//*********************************************************

// No additional StatFlags

//*********************************************************
// UNDI MCast IP to MAC
//*********************************************************

// No additional StatFlags

//*********************************************************
// UNDI NvData
//*********************************************************

// No additional StatFlags

//*********************************************************
// UNDI Get Status
//*********************************************************

// Use to determine if an interrupt has occurred.
#define PXE_STATFLAGS_GET_STATUS_INTERRUPT_MASK        0x000F
#define PXE_STATFLAGS_GET_STATUS_NO_INTERRUPTS         0x0000
```

```
// If set, at least one receive interrupt occurred.
#define PXE_STATFLAGS_GET_STATUS_RECEIVE              0x0001

// If set, at least one transmit interrupt occurred.

#define PXE_STATFLAGS_GET_STATUS_TRANSMIT             0x0002

// If set, at least one command interrupt occurred.
#define PXE_STATFLAGS_GET_STATUS_COMMAND              0x0004

// If set, at least one software interrupt occurred.
#define PXE_STATFLAGS_GET_STATUS_SOFTWARE             0x0008

// This flag is set if the transmitted buffer queue is empty.
// This flag will be set if all transmitted buffer addresses
// get written into the DB.
#define PXE_STATFLAGS_GET_STATUS_TXBUF_QUEUE_EMPTY    0x0010

// This flag is set if no transmitted buffer addresses were
// written into the DB.  (This could be because DBsize was
// too small.)
#define PXE_STATFLAGS_GET_STATUS_NO_TXBUFS_WRITTEN    0x0020

//*******************************************************
// UNDI Fill Header
//*******************************************************

// No additional StatFlags

//*******************************************************
// UNDI Transmit
//*******************************************************

// No additional StatFlags.

//*******************************************************
// UNDI Receive
//*******************************************************

// No additional StatFlags.
```

E.3.4.5 PXE_STATCODE

```
typedef PXE_UINT16 PXE_STATCODE;

#define PXE_STATCODE_INITIALIZE                         0x0000

//*******************************************************
// Common StatCodes returned by all UNDI commands, UNDI protocol
// functions and BC protocol functions.
//*******************************************************

#define PXE_STATCODE_SUCCESS                            0x0000
#define PXE_STATCODE_INVALID_CDB                        0x0001
#define PXE_STATCODE_INVALID_CPB                        0x0002
#define PXE_STATCODE_BUSY                               0x0003
#define PXE_STATCODE_QUEUE_FULL                         0x0004
#define PXE_STATCODE_ALREADY_STARTED                    0x0005
#define PXE_STATCODE_NOT_STARTED                        0x0006
#define PXE_STATCODE_NOT_SHUTDOWN                       0x0007
#define PXE_STATCODE_ALREADY_INITIALIZED                0x0008
#define PXE_STATCODE_NOT_INITIALIZED                    0x0009
#define PXE_STATCODE_DEVICE_FAILURE                     0x000A
#define PXE_STATCODE_NVDATA_FAILURE                     0x000B
#define PXE_STATCODE_UNSUPPORTED                        0x000C
#define PXE_STATCODE_BUFFER_FULL                        0x000D
#define PXE_STATCODE_INVALID_PARAMETER                  0x000E
#define PXE_STATCODE_INVALID_UNDI                       0x000F
#define PXE_STATCODE_IPV4_NOT_SUPPORTED                 0x0010
#define PXE_STATCODE_IPV6_NOT_SUPPORTED                 0x0011
#define PXE_STATCODE_NOT_ENOUGH_MEMORY                  0x0012
#define PXE_STATCODE_NO_DATA                            0x0013
```

E.3.4.6 PXE_IFNUM

```
typedef PXE_UINT16 PXE_IFNUM;

// This interface number must be passed to the S/W UNDI Start
// command.

#define PXE_IFNUM_START                                 0x0000

// This interface number is returned by the S/W UNDI Get State
// and Start commands if information in the CDB, CPB or DB is
// invalid.

#define PXE_IFNUM_INVALID                               0x0000
```

E.3.4.7 PXE_CONTROL

```
typedef PXE_UINT16 PXE_CONTROL;

// Setting this flag directs the UNDI to queue this command for
// later execution if the UNDI is busy and it supports command
// queuing.  If queuing is not supported, a
// PXE_STATCODE_INVALID_CONTROL error is returned.  If the queue
// is full, a PXE_STATCODE_CDB_QUEUE_FULL error is returned.

#define PXE_CONTROL_QUEUE_IF_BUSY                         0x0002

// These two bit values are used to determine if there are more
// UNDI CDB structures following this one.  If the link bit is
// set, there must be a CDB structure following this one.
// Execution will start on the next CDB structure as soon as this
// one completes successfully.  If an error is generated by this
// command, execution will stop.

#define PXE_CONTROL_LINK                                  0x0001
#define PXE_CONTROL_LAST_CDB_IN_LIST                      0x0000
```

E.3.4.8 PXE_FRAME_TYPE

```
typedef PXE_UINT8 PXE_FRAME_TYPE;

#define PXE_FRAME_TYPE_NONE                               0x00
#define PXE_FRAME_TYPE_UNICAST                            0x01
#define PXE_FRAME_TYPE_BROADCAST                          0x02
#define PXE_FRAME_TYPE_FILTERED_MULTICAST                 0x03
#define PXE_FRAME_TYPE_PROMISCUOUS                        0x04
#define PXE_FRAME_TYPE_PROMISCUOUS_MULTICAST              0x05
```

E.3.4.9 PXE_IPV4

This storage type is always big endian (network order) not little endian (Intel order).

```
typedef PXE_UINT32 PXE_IPV4;
```

E.3.4.10 PXE_IPV6

This storage type is always big endian (network order) not little endian (Intel order).

```
typedef struct s_PXE_IPV6 {
  PXE_UINT32 num[4];
} PXE_IPV6;
```

E.3.4.11 PXE_MAC_ADDR

This storage type is always big endian (network order) not little endian (Intel order).

```
typedef struct {
  PXE_UINT8 num[32];
} PXE_MAC_ADDR;
```

E.3.4.12 PXE_IFTYPE

The interface type is returned by the Get Initialization Information command and is used by the BC DHCP protocol function. This field is also used for the low order 8-bits of the H/W type field in ARP packets. The high order 8-bits of the H/W type field in ARP packets will always be set to 0x00 by the BC.

```
typedef PXE_UINT8 PXE_IFTYPE;

// This information is from the ARP section of RFC 1700.

//      1 Ethernet (10Mb)
//      2 Experimental Ethernet (3Mb)
//      3 Amateur Radio AX.25
//      4 Proteon ProNET Token Ring
//      5 Chaos
//      6 IEEE 802 Networks
//      7 ARCNET
//      8 Hyperchannel
//      9 Lanstar
//     10 Autonet Short Address
//     11 LocalTalk
//     12 LocalNet (IBM PCNet or SYTEK LocalNET)
//     13 Ultra link
//     14 SMDS
//     15 Frame Relay
//     16 Asynchronous Transmission Mode (ATM)
//     17 HDLC
//     18 Fibre Channel
//     19 Asynchronous Transmission Mode (ATM)
//     20 Serial Line
//     21 Asynchronous Transmission Mode (ATM)

#define PXE_IFTYPE_ETHERNET                             0x01
#define PXE_IFTYPE_TOKENRING                            0x04
#define PXE_IFTYPE_FIBRE_CHANNEL                        0x12
```

E.3.5 Compound Types

All PXE structures must be byte packed.

E.3.5.1 PXE_HW_UNDI

This section defines the C structures and #defines for the !PXE H/W UNDI interface.

```
#pragma pack(1)
typedef struct s_pxe_hw_undi {
  PXE_UINT32     Signature;       // PXE_ROMID_SIGNATURE
  PXE_UINT8      Len;             // sizeof(PXE_HW_UNDI)
  PXE_UINT8      Fudge;           // makes 8-bit cksum equal zero
  PXE_UINT8      Rev;             // PXE_ROMID_REV
  PXE_UINT8      IFcnt;           // physical connector count
  PXE_UINT8      MajorVer;        // PXE_ROMID_MAJORVER
  PXE_UINT8      MinorVer;        // PXE_ROMID_MINORVER
  PXE_UINT16     reserved;        // zero, not used
  PXE_UINT32     Implementation;  // implementation flags
} PXE_HW_UNDI;
#pragma pack()

// Status port bit definitions

// UNDI operation state

#define PXE_HWSTAT_STATE_MASK                          0xC0000000
#define PXE_HWSTAT_BUSY                                0xC0000000
#define PXE_HWSTAT_INITIALIZED                         0x80000000
#define PXE_HWSTAT_STARTED                             0x40000000
#define PXE_HWSTAT_STOPPED                             0x00000000

// If set, last command failed

#define PXE_HWSTAT_COMMAND_FAILED                      0x20000000

// If set, identifies enabled receive filters

#define PXE_HWSTAT_PROMISCUOUS_MULTICAST_RX_ENABLED    0x00001000
#define PXE_HWSTAT_PROMISCUOUS_RX_ENABLED              0x00000800
#define PXE_HWSTAT_BROADCAST_RX_ENABLED                0x00000400
#define PXE_HWSTAT_MULTICAST_RX_ENABLED                0x00000200
#define PXE_HWSTAT_UNICAST_RX_ENABLED                  0x00000100
```

```
// If set, identifies enabled external interrupts

#define PXE_HWSTAT_SOFTWARE_INT_ENABLED              0x00000080
#define PXE_HWSTAT_TX_COMPLETE_INT_ENABLED           0x00000040
#define PXE_HWSTAT_PACKET_RX_INT_ENABLED             0x00000020
#define PXE_HWSTAT_CMD_COMPLETE_INT_ENABLED          0x00000010

// If set, identifies pending interrupts

#define PXE_HWSTAT_SOFTWARE_INT_PENDING              0x00000008
#define PXE_HWSTAT_TX_COMPLETE_INT_PENDING           0x00000004
#define PXE_HWSTAT_PACKET_RX_INT_PENDING             0x00000002
#define PXE_HWSTAT_CMD_COMPLETE_INT_PENDING          0x00000001

// Command port definitions

// If set, CDB identified in CDBaddr port is given to UNDI.
// If not set, other bits in this word will be processed.

#define PXE_HWCMD_ISSUE_COMMAND                      0x80000000
#define PXE_HWCMD_INTS_AND_FILTS                     0x00000000

// Use these to enable/disable receive filters.

#define PXE_HWCMD_PROMISCUOUS_MULTICAST_RX_ENABLE    0x00001000
#define PXE_HWCMD_PROMISCUOUS_RX_ENABLE              0x00000800
#define PXE_HWCMD_BROADCAST_RX_ENABLE                0x00000400
#define PXE_HWCMD_MULTICAST_RX_ENABLE                0x00000200
#define PXE_HWCMD_UNICAST_RX_ENABLE                  0x00000100

// Use these to enable/disable external interrupts

#define PXE_HWCMD_SOFTWARE_INT_ENABLE                0x00000080
#define PXE_HWCMD_TX_COMPLETE_INT_ENABLE             0x00000040
#define PXE_HWCMD_PACKET_RX_INT_ENABLE               0x00000020
#define PXE_HWCMD_CMD_COMPLETE_INT_ENABLE            0x00000010

// Use these to clear pending external interrupts

#define PXE_HWCMD_CLEAR_SOFTWARE_INT                 0x00000008
#define PXE_HWCMD_CLEAR_TX_COMPLETE_INT              0x00000004
#define PXE_HWCMD_CLEAR_PACKET_RX_INT                0x00000002
#define PXE_HWCMD_CLEAR_CMD_COMPLETE_INT             0x00000001
```

E.3.5.2 PXE_SW_UNDI

This section defines the C structures and #defines for the !PXE S/W UNDI interface.

```
#pragma pack(1)
typedef struct s_pxe_sw_undi {
  PXE_UINT32     Signature;       // PXE_ROMID_SIGNATURE
  PXE_UINT8      Len;             // sizeof(PXE_SW_UNDI)
  PXE_UINT8      Fudge;           // makes 8-bit cksum zero
  PXE_UINT8      Rev;             // PXE_ROMID_REV
  PXE_UINT8      IFcnt;           // physical connector count
  PXE_UINT8      MajorVer;        // PXE_ROMID_MAJORVER
  PXE_UINT8      MinorVer;        // PXE_ROMID_MINORVER
  PXE_UINT16     reserved1;       // zero, not used
  PXE_UINT32     Implementation;  // Implementation flags
  PXE_UINT64     EntryPoint;      // API entry point
  PXE_UINT8      reserved2[3];    // zero, not used
  PXE_UINT8      BusCnt;          // number of bustypes supported
  PXE_UINT32     BusType[1];      // list of supported bustypes
} PXE_SW_UNDI;
#pragma pack()
```

E.3.5.3 PXE_UNDI

PXE_UNDI combines both the H/W and S/W UNDI types into one typedef and has #defines for common fields in both H/W and S/W UNDI types.

```
#pragma pack(1)
typedef union u_pxe_undi {
  PXE_HW_UNDI hw;
  PXE_SW_UNDI sw;
} PXE_UNDI;
#pragma pack()

// Signature of !PXE structure

#define PXE_ROMID_SIGNATURE PXE_BUSTYPE('!', 'P', 'X', 'E')

// !PXE structure format revision

#define PXE_ROMID_REV                                        0x02

// UNDI command interface revision.  These are the values that
// get sent in option 94 (Client Network Interface Identifier) in
// the DHCP Discover and PXE Boot Server Request packets.

#define PXE_ROMID_MAJORVER                                   0x03
#define PXE_ROMID_MINORVER                                   0x01
```

```
// Implementation flags

#define PXE_ROMID_IMP_HW_UNDI                           0x80000000
#define PXE_ROMID_IMP_SW_VIRT_ADDR                      0x40000000
#define PXE_ROMID_IMP_64BIT_DEVICE                      0x00010000
#define PXE_ROMID_IMP_FRAG_SUPPORTED                    0x00008000
#define PXE_ROMID_IMP_CMD_LINK_SUPPORTED                0x00004000
#define PXE_ROMID_IMP_CMD_QUEUE_SUPPORTED               0x00002000
#define PXE_ROMID_IMP_MULTI_FRAME_SUPPORTED             0x00001000
#define PXE_ROMID_IMP_NVDATA_SUPPORT_MASK               0x00000C00
#define PXE_ROMID_IMP_NVDATA_BULK_WRITABLE              0x00000C00
#define PXE_ROMID_IMP_NVDATA_SPARSE_WRITABLE            0x00000800
#define PXE_ROMID_IMP_NVDATA_READ_ONLY                  0x00000400
#define PXE_ROMID_IMP_NVDATA_NOT_AVAILABLE              0x00000000
#define PXE_ROMID_IMP_STATISTICS_SUPPORTED              0x00000200
#define PXE_ROMID_IMP_STATION_ADDR_SETTABLE             0x00000100
#define PXE_ROMID_IMP_PROMISCUOUS_MULTICAST_RX_SUPPORTED \
                                                        0x00000080
#define PXE_ROMID_IMP_PROMISCUOUS_RX_SUPPORTED \        0x00000040
#define PXE_ROMID_IMP_BROADCAST_RX_SUPPORTED \          0x00000020
#define PXE_ROMID_IMP_FILTERED_MULTICAST_RX_SUPPORTED \
                                                        0x00000010
#define PXE_ROMID_IMP_SOFTWARE_INT_SUPPORTED \          0x00000008
#define PXE_ROMID_IMP_TX_COMPLETE_INT_SUPPORTED \       0x00000004
#define PXE_ROMID_IMP_PACKET_RX_INT_SUPPORTED \         0x00000002
#define PXE_ROMID_IMP_CMD_COMPLETE_INT_SUPPORTED \      0x00000001
```

E.3.5.4 PXE_CDB

PXE UNDI command descriptor block.

```
#pragma pack(1)
typedef struct s_pxe_cdb {
 PXE_OPCODE       OpCode;
 PXE_OPFLAGS      OpFlags;
 PXE_UINT16       CPBsize;
 PXE_UINT16       DBsize;
 PXE_UINT64       CPBaddr;
 PXE_UINT64       DBaddr;
 PXE_STATCODE     StatCode;
 PXE_STATFLAGS    StatFlags;
 PXE_UINT16       IFnum;
 PXE_CONTROL      Control;
} PXE_CDB;
#pragma pack()
```

E.3.5.5 PXE_IP_ADDR

This storage type is always big endian (network order) not little endian (Intel order).

```
#pragma pack(1)
typedef union u_pxe_ip_addr {
  PXE_IPV6       IPv6;
  PXE_IPV4       IPv4;
} PXE_IP_ADDR;
#pragma pack()
```

E.3.5.6 PXE_DEVICE

This typedef is used to identify the network device that is being used by the UNDI. This information is returned by the Get Config Info command.

```
#pragma pack(1)
typedef union pxe_device {

// PCI and PC Card NICs are both identified using bus, device
// and function numbers.  For PC Card, this may require PC
// Card services to be loaded in the BIOS or preboot
// environment.
struct {
// See S/W UNDI ROMID structure definition for PCI and
// PCC BusType definitions.
PXE_UINT32       BusType;

// Bus, device & function numbers that locate this device.
PXE_UINT16       Bus;
PXE_UINT8        Device;
PXE_UINT8        Function;
} PCI, PCC;

} PXE_DEVICE;
#pragma pack()
```

E.4 UNDI Commands

All 32/64-bit UNDI commands use the same basic command format, the CDB (Command Descriptor Block). CDB fields that are not used by a particular command must be initialized to zero by the application/driver that is issuing the command.

All UNDI implementations must set the command completion status (**PXE_STATFLAGS_COMMAND_COMPLETE**) after command execution completes. Applications and drivers must not alter or rely on the contents of any of the CDB, CPB or DB fields until the command completion status is set.

All commands return status codes for invalid CDB contents and, if used, invalid CPB contents. Commands with invalid parameters will not execute. Fix the error and submit the command again.

Figure E-6 describes the different UNDI states (Stopped, Started and Initialized), shows the transitions between the states and which UNDI commands are valid in each state.

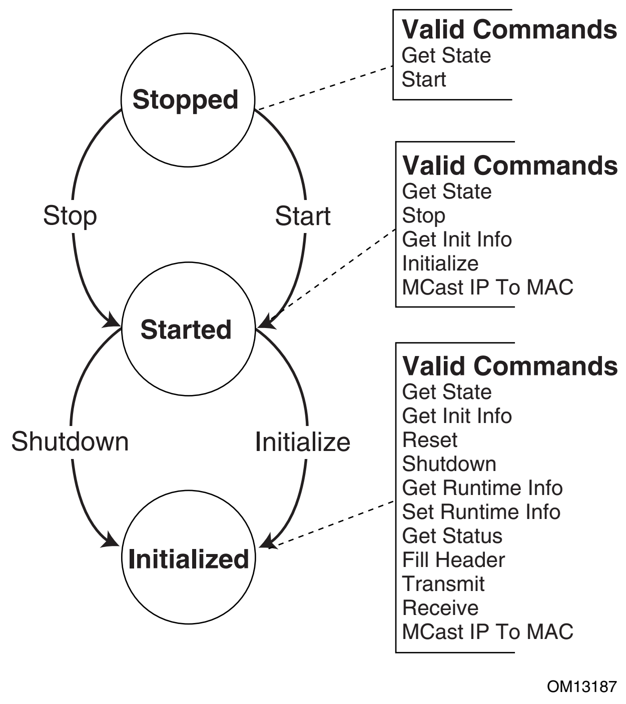


Figure E-6. UNDI States, Transitions & Valid Commands

NOTE

All memory addresses including the CDB address, CPB address, and the DB address submitted to the S/W UNDI by the protocol drivers must be processor-based addresses. All memory addresses submitted to the H/W UNDI must be device based addresses.

NOTE

Additional requirements for S/W UNDI implementations: Processor register contents must be unchanged by S/W UNDI command execution (the application/driver does not have to save processor registers when calling S/W UNDI). Processor arithmetic flags are undefined (application/driver must save processor arithmetic flags if needed). Application/driver must remove CDB address from stack after control returns from S/W UNDI.

NOTE

Additional requirements for 32-bit network devices: All addresses given to the S/W UNDI must be 32-bit addresses. Any address that exceeds 32 bits (4 GB) will result in a return of one of the following status codes: PXE_STATCODE_INVALID_PARAMETER, PXE_STATCODE_INVALID_CDB or PXE_STATCODE_INVALID_CPB.

When executing linked commands, command execution will stop at the end of the CDB list (when the **`PXE_CONTROL_LINK`** bit is not set) or when a command returns an error status code.

E.4.1 Command Linking and Queuing

When linking commands, the CDBs must be stored consecutively in system memory without any gaps in between. Do not set the Link bit in the last CDB in the list. As shown in Figure E-7, the Link bit must be set in all other CDBs in the list.

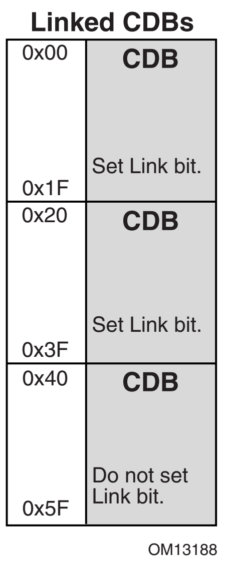


Figure E-7. Linked CDBs

When the H/W UNDI is executing commands, the State bits in the Status field in the !PXE structure will be set to Busy (3).

When H/W or S/W UNDI is executing commands and a new command is issued, a StatCode of **PXE_STATCODE_BUSY** and a StatFlag of **PXE_STATFLAG_COMMAND_FAILURE** is set in the CDB. For linked commands, only the first CDB will be set to Busy, all other CDBs will be unchanged. When a linked command fails, execution on the list stops. Commands after the failing command will not be run.

As shown in Figure E-8, when queuing commands, only the first CDB needs to have the Queue Control flag set. If queuing is supported and the UNDI is busy and there is room in the command queue, the command (or list of commands) will be queued.

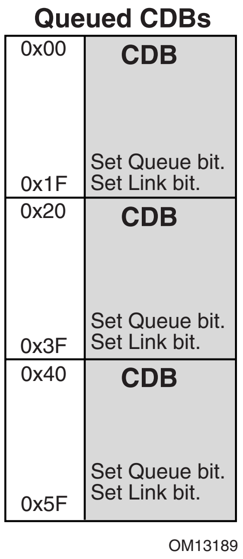


Figure E-8. Queued CDBs

When a command is queued a StatFlag of **PXE_STATFLAG_COMMAND_QUEUED** is set (if linked commands are queued only the StatFlag of the first CDB gets set). This signals that the command was added to the queue. Commands in the queue will be run on a first-in, first-out, basis. When a command fails, the next command in the queue is run. When a linked command in the queue fails, execution on the list stops. The next command, or list of commands, that was added to the command queue will be run.

E.4.2 Get State

This command is used to determine the operational state of the UNDI. An UNDI has three possible operational states:

Stopped: A stopped UNDI is free for the taking. When all interface numbers (IFnum) for a particular S/W UNDI are stopped, that S/W UNDI image can be relocated or removed. A stopped UNDI will accept Get State and Start commands.

Started: A started UNDI is in use. A started UNDI will accept Get State, Stop, Get Init Info, and Initialize commands.

Initialized: An initialized UNDI is in used. An initialized UNDI will accept all commands except: Start, Stop, and Initialize.

Drivers, NBPs, and applications should not use UNDIs that are already started or initialized.

No other operational checks are made by this command. If this is a S/W UNDI, the `PXE_START_CPB.Delay()` and `PXE_START_CPB.Virt2Phys()` callbacks will not be used.

E.4.2.1 Issuing the Command

To issue a Get State command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a Get State command
OpCode	`PXE_OPCODE_GET_STATE`
OpFlags	`PXE_OPFLAGS_NOT_USED`
CPBsize	`PXE_CPBSIZE_NOT_USED`
DBsize	`PXE_DBSIZE_NOT_USED`
CPBaddr	`PXE_CPBADDR_NOT_USED`
DBaddr	`PXE_DBADDR_NOT_USED`
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`
Control	Set as needed

E.4.2.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. StatFlags contain operational state.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued. All other fields are unchanged.
INITIALIZE	Command has not been executed or queued.

E.4.2.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. StatFlags contain operational state.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.

If the command completes successfully, use `PXE_STATFLAGS_GET_STATE_MASK` to check the state of the UNDI.

StatFlags	Reason
STOPPED	The UNDI is stopped.
STARTED	The UNDI is started, but not initialized.
INITIALIZED	The UNDI is initialized.

E.4.3 Start

This command is used to change the UNDI operational state from stopped to started. No other operational checks are made by this command. Protocol driver makes this call for each network interface supported by the UNDI with a set of call back routines and a unique identifier to identify the particular interface. UNDI does not interpret the unique identifier in any way except that it is a 64-bit value and it will pass it back to the protocol driver as a parameter to all the call back routines for any particular interface. If this is a S/W UNDI, the callback functions Delay(), Virt2Phys(), Map_Mem(), UnMap_Mem(), and Sync_Mem() functions will not be called by this command.

E.4.3.1 Issuing the Command

To issue a Start command for H/W UNDI, create a CDB and fill it in as shows in the table below:

CDB Field	How to initialize the CDB structure for a H/W UNDI Start command
OpCode	`PXE_OPCODE_START`
OpFlags	`PXE_OPFLAGS_NOT_USED`
CPBsize	`PXE_CPBSIZE_NOT_USED`
DBsize	`PXE_DBSIZE_NOT_USED`
CPBaddr	`PXE_CPBADDR_NOT_USED`
DBaddr	`PXE_DBADDR_NOT_USED`
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`
Control	Set as needed

To issue a Start command for S/W UNDI, create a CDB and fill it in as shows in the table below:

CDB Field	How to initialize the CDB structure for a S/W UNDI Start command
OpCode	`PXE_OPCODE_START`
OpFlags	`PXE_OPFLAGS_NOT_USED`
CPBsize	`sizeof(PXE_CPB_START)`
DBsize	`PXE_DBSIZE_NOT_USED`
CPBaddr	Address of a `PXE_CPB_START` structure.
DBaddr	`PXE_DBADDR_NOT_USED`
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`
Control	Set as needed

Preparing the CPB

The CPB for the S/W UNDI Start command (shown below) must be filled in and the size and address of the CPB must be given in the CDB.

```
#pragma pack(1)
typedef struct s_pxe_cpb_start {
  // PXE_VOID Delay(PXE_UINT64 microseconds);

  // UNDI will never request a delay smaller than 10 microseconds
  // and will always request delays in increments of 10
  // microseconds.  The Delay() CallBack routine must delay
  // between n and n + 10 microseconds before returning control
  // to the UNDI.

  // This field cannot be set to zero.
  PXE_UINT64     Delay;

  // PXE_VOID Block(PXE_UINT32 enable);

  // UNDI may need to block multithreaded/multiprocessor access
  // to critical code sections when programming or accessing the
  // network device.  To this end, a blocking service is needed
  // by the UNDI.  When UNDI needs a block, it will call Block()
  // passing a nonzero value.  When UNDI no longer needs a
  // block, it will call Block() with a zero value.  When called,
  // if the Block() is already enabled, do not return control to
  // the UNDI until the previous Block() is disabled.

  // This field cannot be set to zero.
  PXE_UINT64     Block;

  // PXE_VOID Virt2Phys(PXE_UINT64 virtual, PXE_UINT64
  // physical_ptr);

  // UNDI will pass the virtual address of a buffer and the
  // virtual address of a 64-bit physical buffer.  Convert the
  // virtual address to a physical address and write the result
  // to the physical address buffer.  If virtual and physical
  // addresses are the same, just copy the virtual address to the
  // physical address buffer.

  // This field can be set to zero if virtual and physical
  // addresses are equal.

  PXE_UINT64     Virt2Phys;

  // PXE_VOID Mem_IO(UINT64 unq_id, PXE_UINT8 read_write,
  //PXE_UINT8 len, PXE_UINT64 port, PXE_UINT64 buf_addr);
  //
```

```
// UNDI will read or write the device io space using this
// call-back function. It passes the number of bytes as the
// len parameter and it will be either 1,2,4 or 8.
//
// This field cannot be set to zero.
//

PXE_UINT64      Mem_IO;

//
// PXE_VOID Map_Mem(UINT64 unq_id, UINT64 virtual_addr,
// UINT32 size, UINT32 Direction, UINT64 mapped_addr);
//
// UNDI will pass the virtual address of a buffer, direction of
// the data flow from/to the mapped buffer (the constants
// values for the direction flag are defined below)
// and a place holder (pointer) for the mapped address.
// This call will Map the given address to a physical DMA
// address and write the result to the mapped_addr pointer.  If
// there is no need to map the given address to a lower
// physical address i.e. the given virtual address points to a
// physical memory that can be used for the DMA read/write, it
// converts the given virtual address to its physical address
// and write that in the mapped address pointer.
//
// This field can be set to zero if there is no mapping service
// available or if all the physical addresses are DMA
// compatible.
//
UINT64    Map_Mem;

//
// PXE_VOID UnMap_Mem(UINT64 unq_id, UINT64 virtual_addr,
// UINT32 size,     UINT32 Direction, UINT64 mapped_addr);
//
// UNDI will pass the virtual and mapped addresses of a buffer
// This call will un map the given address
//
// This field can be set to zero if there is no unmapping
// service available
//
UINT64    UnMap_Mem;
```

```
    //
    // PXE_VOID Sync_Mem(UINT64 unq_id, UINT64 virtual,
    //  UINT32 size, UINT32 Direction, UINT64 mapped_addr);
    //
    // UNDI will pass the virtual and mapped addresses of a buffer
    // This call will synchronize the contents of both the virtual
    // and mapped buffers for the given Direction. This call does
    // not do anything if both the virtual and mapped addresses
    // point to the same physical memory.
    //
    // This field can be set to zero if there is no service
    // available
    //
    UINT64   Sync_Mem;

    // protocol driver can provide anything for this Unique_ID,
    // UNDI remembers that as just a 64bit value assocaited to the
    // interface specified by the ifnum and gives it back as a
    // parameter to all the call-back routines when calling for
    // that interface!

    UINT64   Unique_ID;

  } PXE_CPB_START;
  #pragma pack()

  #define TO_AND_FROM_DEVICE                     0
  #define FROM_DEVICE                            1
  #define TO_DEVICE                              2
  #define PXE_DELAY_MILLISECOND                  1000
  #define PXE_DELAY_SECOND                       1000000
  #define PXE_IO_READ                            0
  #define PXE_IO_WRITE                           1
  #define PXE_MEM_READ                           2
  #define PXE_MEM_WRITE                          4
```

E.4.3.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the **CDB.StatFlags** field. Until these bits change to report **PXE_STATFLAGS_COMMAND_COMPLETE** or **PXE_STATFLAGS_COMMAND_FAILED**, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. UNDI is now started.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.3.3 Checking Command Execution Results

After command execution completes, either successfully or not, the **CDB.StatCode** field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. UNDI is now started.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
ALREADY_STARTED	The UNDI is already started.

E.4.4 Stop

This command is used to change the UNDI operational state from started to stopped.

E.4.4.1 Issuing the Command

To issue a Stop command, create a CDB and fill it in as shows in the table below:

CDB Field	How to initialize the CDB structure for a Stop command
OpCode	`PXE_OPCODE_STOP`
OpFlags	`PXE_OPFLAGS_NOT_USED`
CPBsize	`PXE_CPBSIZE_NOT_USED`
DBsize	`PXE_DBSIZE_NOT_USED`
CPBaddr	`PXE_CPBADDR_NOT_USED`
DBaddr	`PXE_DBADDR_NOT_USED`
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`
Control	Set as needed

E.4.4.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. UNDI is now stopped.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has not been executed or queued.

E.4.4.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. UNDI is now stopped.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_SHUTDOWN	The UNDI is initialized and must be shutdown before it can be stopped.

E.4.5 Get Init Info

This command is used to retrieve initialization information that is needed by drivers and applications to initialized UNDI.

E.4.5.1 Issuing the Command

To issue a Get Init Info command, create a CDB and fill it in as shows in the table below:

CDB Field	How to initialize the CDB structure for a Get Init Info command
OpCode	`PXE_OPCODE_GET_INIT_INFO`
OpFlags	`PXE_OPFLAGS_NOT_USED`
CPBsize	`PXE_CPBSIZE_NOT_USED`
DBsize	`sizeof(PXE_DB_INIT_INFO)`
CPBaddr	`PXE_CPBADDR_NOT_USED`
DBaddr	Address of a `PXE_DB_INIT_INFO` structure.
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

E.4.5.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. DB can be used.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.5.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. DB can be used.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.

StatFlags

To determine if cable detection is supported by this UNDI/NIC, use these macros with the value returned in the CDB.StatFlags field:

```
PXE_STATFLAGS_CABLE_DETECT_MASK
PXE_STATFLAGS_CABLE_DETECT_NOT_SUPPORTED
PXE_STATFLAGS_CABLE_DETECT_SUPPORTED
```

DB

```
#pragma pack(1)
typedef struct s_pxe_db_get_init_info {

  // Minimum length of locked memory buffer that must be given to
  // the Initialize command.  Giving UNDI more memory will
  // generally give better performance.

  // If MemoryRequired is zero, the UNDI does not need and will
  // not use system memory to receive and transmit packets.

  PXE_UINT32     MemoryRequired;

  // Maximum frame data length for Tx/Rx excluding the media
  // header.
  //
  PXE_UINT32     FrameDataLen;

  // Supported link speeds are in units of mega bits.  Common
  // ethernet values are 10, 100 and 1000.  Unused LinkSpeeds[]
  // entries are zero filled.

  PXE_UINT32     LinkSpeeds[4];

  // Number of nonvolatile storage items.

  PXE_UINT32     NvCount;

  // Width of nonvolatile storage item in bytes.  0, 1, 2 or 4

  PXE_UINT16     NvWidth;
```

```
// Media header length.  This is the typical media header
// length for this UNDI.  This information is needed when
// allocating receive and transmit buffers.

PXE_UINT16     MediaHeaderLen;

// Number of bytes in the NIC hardware (MAC) address.

PXE_UINT16     HWaddrLen;

// Maximum number of multicast MAC addresses in the multicast
// MAC address filter list.

PXE_UINT16     MCastFilterCnt;

// Default number and size of transmit and receive buffers that
// will be allocated by the UNDI.  If MemoryRequired is
// nonzero, this allocation will come out of the memory buffer
// given to the Initialize command.  If MemoryRequired is zero,
// this allocation will come out of memory on the NIC.

PXE_UINT16     TxBufCnt;
PXE_UINT16     TxBufSize;
PXE_UINT16     RxBufCnt;
PXE_UINT16     RxBufSize;

// Hardware interface types defined in the Assigned Numbers RFC
// and used in DHCP and ARP packets.
// See the PXE_IFTYPE typedef and PXE_IFTYPE_xxx macros.

PXE_UINT8      IFtype;

// Supported duplex options.  This can be one or a combination
// of more than one constants defined as PXE_DUPLEX_xxxxx
// below. This value indicates the ability of UNDI to
// change/control the duplex modes of the NIC.

PXE_UINT8      SupportedDuplexModes;
```

```
  // Supported loopback options. This field can be one or a
  // combination of more than one constants defined as
  // PXE_LOOPBACK_xxxxx #defines  below. This value indicates
  // the ability of UNDI to change/control the loopback modes
  // of the NIC

  PXE_UINT8      SupportedLoopBackModes;
} PXE_DB_GET_INIT_INFO;
#pragma pack()

#define PXE_MAX_TXRX_UNIT_ETHER                             1500
#define PXE_HWADDR_LEN_ETHER                                0x0006

#define PXE_DUPLEX_AUTO_DETECT_SUPPORTED                    1
define PXE_DUPLEX_FORCE_FULL_SUPPORTED                      2
#define PXE_DUPLEX_FORCE_HALF_SUPPORTED                     4

#define PXE_LOOPBACK_INTERNAL_SUPPORTED                     1
#define PXE_LOOPBACK_EXTERNAL_SUPPORTED                     2
```

E.4.6 Get Config Info

This command is used to retrieve configuration information about the NIC being controlled by the UNDI.

E.4.6.1 Issuing the Command

To issue a Get Config Info command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a Get Config Info command
OpCode	`PXE_OPCODE_GET_CONFIG_INFO`
OpFlags	`PXE_OPFLAGS_NOT_USED`
CPBsize	`PXE_CPBSIZE_NOT_USED`
DBsize	`sizeof(PXE_DB_CONFIG_INFO)`
CPBaddr	`PXE_CPBADDR_NOT_USED`
DBaddr	Address of a `PXE_DB_CONFIG_INFO` structure
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`
Control	Set as needed

E.4.6.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. DB has been written.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.6.3 Checking Command Execution Results

After command execution completes, either successfully or not, the **CDB.StatCode** field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. DB has been written.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.

DB

```
#pragma pack(1)
typedef struct s_pxe_pci_config_info {

  // This is the flag field for the PXE_DB_GET_CONFIG_INFO union.
  // For PCI bus devices, this field is set to PXE_BUSTYPE_PCI.

  PXE_UINT32     BusType;

  // This identifies the PCI network device that this UNDI
  // interface is bound to.

  PXE_UINT16     Bus;
  PXE_UINT8      Device;
  PXE_UINT8      Function;

  // This is a copy of the PCI configuration space for this
  // network device.

  union {
      PXE_UINT8        Byte[256];
      PXE_UINT16       Word[128];
      PXE_UINT32       Dword[64];
  } Config;
} PXE_PCI_CONFIG_INFO;
#pragma pack()
#pragma pack(1)
typedef struct s_pxe_pcc_config_info {
  // This is the flag field for the PXE_DB_GET_CONFIG_INFO union.
  // For PCC bus devices, this field is set to PXE_BUSTYPE_PCC.

  PXE_UINT32      BusType;

  // This identifies the PCC network device that this UNDI
  // interface is bound to.

  PXE_UINT16      Bus;
  PXE_UINT8       Device;
  PXE_UINT8       Function;

  // This is a copy of the PCC configuration space for this
  // network device.

  union {
      PXE_UINT8           Byte[256];
      PXE_UINT16          Word[128];
      PXE_UINT32          Dword[64];
} Config;
} PXE_PCC_CONFIG_INFO;
#pragma pack()

#pragma pack(1)
typedef union u_pxe_db_get_config_info {
  PXE_PCI_CONFIG_INFO         pci;
  PXE_PCC_CONFIG_INFO         pcc;
} PXE_DB_GET_CONFIG_INFO;
#pragma pack()
```

E.4.7 Initialize

This command resets the network adapter and initializes UNDI using the parameters supplied in the CPB. The Initialize command must be issued before the network adapter can be setup to transmit and receive packets. This command will not enable the receive unit or external interrupts.

Once the memory requirements of the UNDI are obtained by using the Get Init Info command, a block of kernel (nonswappable) memory may need to be allocated by the protocol driver. The address of this kernel memory must be passed to UNDI using the Initialize command CPB. This memory is used for transmit and receive buffers and internal processing.

Initializing the network device will take up to four seconds for most network devices and in some extreme cases (usually poor cables) up to twenty seconds. Control will not be returned to the caller and the **COMMAND_COMPLETE** status flag will not be set until the NIC is ready to transmit.

E.4.7.1 Issuing the Command

To issue an Initialize command, create a CDB and fill it in as shows in the table below:

CDB Field	How to initialize the CDB structure for an Initialize command
OpCode	**PXE_OPCODE_INITIALIZE**
OpFlags	Set as needed.
CPBsize	**sizeof(PXE_CPB_INITIALIZE)**
DBsize	**sizeof(PXE_DB_INITIALIZE)**
CPBaddr	Address of a **PXE_CPB_INITIALIZE** structure.
Dbaddr	Address of a **PXE_DB_INITIALIZE** structure.
StatCode	**PXE_STATCODE_INITIALIZE**
StatFlags	**PXE_STATFLAGS_INITIALIZE**
Ifnum	A valid interface number from zero to **!PXE.IFcnt**.
Control	Set as needed.

OpFlags

Cable detection can be enabled or disabled by setting one of the following OpFlags:

PXE_OPFLAGS_INITIALIZE_CABLE_DETECT
PXE_OPFLAGS_INITIALIZE_DO_NOT_DETECT_CABLE

Preparing the CPB

If the `MemoryRequired` field returned in the `PXE_DB_GET_INIT_INFO` structure is zero, the Initialize command does not need to be given a memory buffer or even a CPB structure. If the `MemoryRequired` field is nonzero, the Initialize command does need a memory buffer.

```
#pragma pack(1)
typedef struct s_pxe_cpb_initialize {

  // Address of first (lowest) byte of the memory buffer.
  // This buffer must be in contiguous physical memory and cannot
  // be swapped out.  The UNDI  will be using this for transmit
  // and receive buffering.  This address must be a processor-
  // based address for S/W UNDI and a device-based address for
  // H/W UNDI.

  PXE_UINT64     MemoryAddr;

  // MemoryLength must be greater than or equal to MemoryRequired
  // returned by the Get Init Info command.

  PXE_UINT32     MemoryLength;

  // Desired link speed in Mbit/sec.  Common ethernet values are
  // 10, 100 and 1000.  Setting a value of zero will auto-detect
  // and/or use the default link speed (operation depends on
  // UNDI/NIC functionality).

  PXE_UINT32     LinkSpeed;

  // Suggested number and size of receive and transmit buffers to
  // allocate.  If MemoryAddr and MemoryLength are nonzero, this
  // allocation comes out of the supplied memory buffer.  If
  // MemoryAddr and MemoryLength are zero, this allocation comes
  // out of memory on the NIC.

  // If these fields are set to zero, the UNDI will allocate
  // buffer counts and sizes as it sees fit.

  PXE_UINT16     TxBufCnt;
  PXE_UINT16     TxBufSize;
  PXE_UINT16     RxBufCnt;
  PXE_UINT16     RxBufSize;
```

```
    // The following configuration parameters are optional and must
    // be zero  to use the default values.
    // The possible values for these parameters are defined below.

    PXE_UINT8       DuplexMode;

    PXE_UINT8       LoopBackMode;
  } PXE_CPB_INITIALIZE;
  #pragma pack()

  #define PXE_DUPLEX_AUTO_DETECT          0x00
  #define PXE_FORCE_FULL_DUPLEX           0x01

  #define PXE_FORCE_HALF_DUPLEX           0x02

  #define PXE_LOOPBACK_NORMAL             0
  #define PXE_LOOPBACK_INTERNAL           1
  #define PXE_LOOPBACK_EXTERNAL           2
```

E.4.7.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the **CDB.StatFlags** field. Until these bits change to report **PXE_STATFLAGS_COMMAND_COMPLETE** or **PXE_STATFLAGS_COMMAND_FAILED**, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. UNDI and network device is now initialized. DB has been written.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.7.3 Checking Command Execution Results

After command execution completes, either successfully or not, the **CDB.StatCode** field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. UNDI and network device is now initialized. DB has been written. Check StatFlags.
INVALID_CDB	One of the CDB fields was not set correctly.
INVALID_CPB	One of the CPB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
ALREADY_INITIALIZED	The UNDI is already initialized.
DEVICE_FAILURE	The network device could not be initialized.
NVDATA_FAILURE	The nonvolatile storage could not be read.

StatFlags

Check the StatFlags to see if there is an active connection to this network device. If the no media StatFlag is set, the UNDI and network device are still initialized.

PXE_STATFLAGS_INITIALIZED_NO_MEDIA

Before Using the DB

```
#pragma pack(1)
typedef struct s_pxe_db_initialize {

  // Actual amount of memory used from the supplied memory
  // buffer.  This may be less that the amount of memory
  // supplied and may be zero if the UNDI and network device
  // do not use external memory buffers.  Memory used by the
  // UNDI and network device is allocated from the lowest
  // memory buffer address.

  PXE_UINT32     MemoryUsed;

  // Actual number and size of receive and transmit buffers that
  // were allocated.

  PXE_UINT16     TxBufCnt;
  PXE_UINT16     TxBufSize;
  PXE_UINT16     RxBufCnt;
  PXE_UINT16     RxBufSize
} PXE_DB_INITIALIZE;
#pragma pack()
```

E.4.8 Reset

This command resets the network adapter and reinitializes the UNDI with the same parameters provided in the Initialize command. The transmit and receive queues are emptied and any pending interrupts are cleared. Depending on the state of the OpFlags, the receive filters and external interrupt enables may also be reset.

Resetting the network device may take up to four seconds and in some extreme cases (usually poor cables) up to twenty seconds. Control will not be returned to the caller and the `COMMAND_COMPLETE` status flag will not be set until the NIC is ready to transmit.

E.4.8.1 Issuing the Command

To issue a Reset command, create a CDB and fill it in as shows in the table below:

CDB Field	How to initialize the CDB structure for a Reset command
OpCode	`PXE_OPCODE_RESET`
OpFlags	Set as needed.
CPBsize	`PXE_CPBSIZE_NOT_USED`
DBsize	`PXE_DBSIZE_NOT_USED`
CPBaddr	`PXE_CPBSIZE_NOT_USED`
DBaddr	`PXE_DBSIZE_NOT_USED`
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

OpFlags

Normally the settings of the receive filters and external interrupt enables are unchanged by the Reset command. These two OpFlags will alter the operation of the Reset command.

```
PXE_OPFLAGS_RESET_DISABLE_INTERRUPTS
PXE_OPFLAGS_RESET_DISABLE_FILTERS
```

E.4.8.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the **CDB.StatFlags** field. Until these bits change to report **PXE_STATFLAGS_COMMAND_COMPLETE** or **PXE_STATFLAGS_COMMAND_FAILED**, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. UNDI and network device have been reset. Check StatFlags.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.8.3 Checking Command Execution Results

After command execution completes, either successfully or not, the **CDB.StatCode** field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. UNDI and network device have been reset. Check StatFlags.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.
DEVICE_FAILURE	The network device could not be initialized.
NVDATA_FAILURE	The nonvolatile storage is not valid.

StatFlags

Check the StatFlags to see if there is an active connection to this network device. If the no media StatFlag is set, the UNDI and network device are still reset.

PXE_STATFLAGS_RESET_NO_MEDIA

E.4.9 Shutdown

The Shutdown command resets the network adapter and leaves it in a safe state for another driver to initialize. Any pending transmits or receives are lost. Receive filters and external interrupt enables are reset (disabled). The memory buffer assigned in the Initialize command can be released or reassigned.

Once UNDI has been shutdown, it can then be stopped or initialized again. The Shutdown command changes the UNDI operational state from initialized to started.

E.4.9.1 Issuing the Command

To issue a Shutdown command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a Shutdown command
OpCode	`PXE_OPCODE_SHUTDOWN`
OpFlags	`PXE_OPFLAGS_NOT_USED`
CPBsize	`PXE_CPBSIZE_NOT_USED`
DBsize	`PXE_DBSIZE_NOT_USED`
CPBaddr	`PXE_CPBSIZE_NOT_USED`
DBaddr	`PXE_DBSIZE_NOT_USED`
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

E.4.9.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the **CDB.StatFlags** field. Until these bits change to report **PXE_STATFLAGS_COMMAND_COMPLETE** or **PXE_STATFLAGS_COMMAND_FAILED**, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. UNDI and network device are shutdown.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.9.3 Checking Command Execution Results

After command execution completes, either successfully or not, the **CDB.StatCode** field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. UNDI and network device are shutdown.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.

E.4.10 Interrupt Enables

The Interrupt Enables command can be used to read and/or change the current external interrupt enable settings. Disabling an external interrupt enable prevents an external (hardware) interrupt from being signaled by the network device, internally the interrupt events can still be polled by using the Get Status command.

E.4.10.1 Issuing the Command

To issue an Interrupt Enables command, create a CDB and fill it in as shows in the table below:

CDB Field	How to initialize the CDB structure for an Interrupt Enables command
OpCode	`PXE_OPCODE_INTERRUPT_ENABLES`
OpFlags	Set as needed.
CPBsize	`PXE_CPBSIZE_NOT_USED`
DBsize	`PXE_DBSIZE_NOT_USED`
CPBaddr	`PXE_CPBADDR_NOT_USED`
DBaddr	`PXE_DBADDR_NOT_USED`
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

OpFlags

To read the current external interrupt enables settings set `CDB.OpFlags` to:

- `PXE_OPFLAGS_INTERRUPT_READ`

To enable or disable external interrupts set one of these OpFlags:

- `PXE_OPFLAGS_INTERRUPT_DISABLE`
- `PXE_OPFLAGS_INTERRUPT_ENABLE`

When enabling or disabling interrupt settings, the following additional OpFlag bits are used to specify which types of external interrupts are to be enabled or disabled:

- `PXE_OPFLAGS_INTERRUPT_RECEIVE`
- `PXE_OPFLAGS_INTERRUPT_TRANSMIT`
- `PXE_OPFLAGS_INTERRUPT_COMMAND`
- `PXE_OPFLAGS_INTERRUPT_SOFTWARE`

Setting `PXE_OPFLAGS_INTERRUPT_SOFTWARE` does not enable an external interrupt type, it generates an external interrupt.

E.4.10.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the **CDB.StatFlags** field. Until these bits change to report **PXE_STATFLAGS_COMMAND_COMPLETE** or **PXE_STATFLAGS_COMMAND_FAILED**, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. Check StatFlags.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.10.3 Checking Command Execution Results

After command execution completes, either successfully or not, the **CDB.StatCode** field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. Check StatFlags.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.

StatFlags

If the command was successful, the **CDB.StatFlags** field reports which external interrupt enable types are currently set. Possible **CDB.StatFlags** bit settings are:

- **PXE_STATFLAGS_INTERRUPT_RECEIVE**
- **PXE_STATFLAGS_INTERRUPT_TRANSMIT**
- **PXE_STATFLAGS_INTERRUPT_COMMAND**

The bits set in **CDB.StatFlags** may be different than those that were requested in **CDB.OpFlags**. For example: If transmit and receive share an external interrupt line, setting either the transmit or receive interrupt will always enable both transmit and receive interrupts. In this case both transmit and receive interrupts will be reported in **CDB.StatFlags**. Always expect to get more than you ask for!

E.4.11 Receive Filters

This command is used to read and change receive filters and, if supported, read and change the multicast MAC address filter list. Control will not be returned to the caller and the `COMMAND_COMPLETE` status flag will not be set until the NIC is ready to receive.

E.4.11.1 Issuing the Command

To issue a Receive Filters command, create a CDB and fill it in as shows in the table below:

CDB Field	How to initialize the CDB structure for a Receive Filters command
OpCode	`PXE_OPCODE_RECEIVE_FILTERS`
OpFlags	Set as needed.
CPBsize	`sizeof(PXE_CPB_RECEIVE_FILTERS)`
DBsize	`sizeof(PXE_DB_RECEIVE_FILTERS)`
CPBaddr	Address of `PXE_CPB_RECEIVE_FILTERS` structure.
DBaddr	Address of `PXE_DB_RECEIVE_FILTERS` structure.
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

OpFlags

To read the current receive filter settings set the `CDB.OpFlags` field to:

- `PXE_OPFLAGS_RECEIVE_FILTER_READ`

To change the current receive filter settings set one of these OpFlag bits:

- `PXE_OPFLAGS_RECEIVE_FILTER_ENABLE`
- `PXE_OPFLAGS_RECEIVE_FILTER_DISABLE`

When changing the receive filter settings, at least one of the OpFlag bits in this list must be selected:

- `PXE_OPFLAGS_RECEIVE_FILTER_UNICAST`
- `PXE_OPFLAGS_RECEIVE_FILTER_BROADCAST`
- `PXE_OPFLAGS_RECEIVE_FILTER_FILTERED_MULTICAST`
- `PXE_OPFLAGS_RECEIVE_FILTER_PROMISCUOUS`
- `PXE_OPFLAGS_RECEIVE_FILTER_ALL_MULTICAST`

To clear the contents of the multicast MAC address filter list, set this OpFlag:

- `PXE_OPFLAGS_RECEIVE_FILTER_RESET_MCAST_LIS`T

Preparing the CPB

The receive filter CPB is used to change the contents multicast MAC address filter list. To leave the multicast MAC address filter list unchanged, set the `CDB.CPBsize` field to `PXE_CPBSIZE_NOT_USED` and `CDB.CPBaddr` to `PXE_CPBADDR_NOT_USED`.

To change the multicast MAC address filter list, set `CDB.CPBsize` to the size, in bytes, of the multicast MAC address filter list and set `CDB.CPBaddr` to the address of the first entry in the multicast MAC address filter list.

```
typedef struct s_pxe_cpb_receive_filters {

  // List of multicast MAC addresses.  This list, if present,
  // will replace the existing multicast MAC address filter list.

  PXE_MAC_ADDR   MCastList[n];
} PXE_CPB_RECEIVE_FILTERS;
```

E.4.11.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. Check StatFlags. DB is written.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.11.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. Check StatFlags. DB is written.
INVALID_CDB	One of the CDB fields was not set correctly.
INVALID_CPB	One of the CPB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.

StatFlags

The receive filter settings in CDB.StatFlags are:

- `PXE_STATFLAGS_RECEIVE_FILTER_UNICAST`
- `PXE_STATFLAGS_RECEIVE_FILTER_BROADCAST`
- `PXE_STATFLAGS_RECEIVE_FILTER_FILTERED_MULTICAST`
- `PXE_STATFLAGS_RECEIVE_FILTER_PROMISCUOUS`
- `PXE_STATFLAGS_RECEIVE_FILTER_ALL_MULTICAST`

Unsupported receive filter settings in OpFlags are promoted to the next more liberal receive filter setting. For example: If broadcast or filtered multicast are requested and are not supported by the network device, but promiscuous is; the promiscuous status flag will be set.

DB

The DB is used to read the current multicast MAC address filter list. The CDB.DBsize and CDB.DBaddr fields can be set to PXE_DBSIZE_NOT_USED and PXE_DBADDR_NOT_USED if the multicast MAC address filter list does not need to be read. When reading the multicast MAC address filter list extra entries in the DB will be filled with zero.

```
typedef struct s_pxe_db_receive_filters {

  // Filtered multicast MAC address list.

  PXE_MAC_ADDR   MCastList[n];
} PXE_DB_RECEIVE_FILTERS;
```

E.4.12 Station Address

This command is used to get current station and broadcast MAC addresses and, if supported, to change the current station MAC address.

E.4.12.1 Issuing the Command

To issue a Station Address command, create a CDB and fill it in as shows in the table below:

CDB Field	How to initialize the CDB structure for a Station Address command
OpCode	`PXE_OPCODE_STATION_ADDRESS`
OpFlags	Set as needed.
CPBsize	`sizeof(PXE_CPB_STATION_ADDRESS)`
DBsize	`sizeof(PXE_DB_STATION_ADDRESS)`
CPBaddr	Address of `PXE_CPB_STATION_ADDRESS` structure.
DBaddr	Address of `PXE_DB_STATION_ADDRESS` structure.
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

OpFlags

To read current station and broadcast MAC addresses set the OpFlags field to:

- `PXE_OPFLAGS_STATION_ADDRESS_READ`

To change the current station to the address given in the CPB set the OpFlags field to:

- `PXE_OPFLAGS_STATION_ADDRESS_WRITE`

To reset the current station address back to the power on default, set the OpFlags field to:

- `PXE_OPFLAGS_STATION_ADDRESS_RESET`

Preparing the CPB

To change the current station MAC address the `CDB.CPBsize` and `CDB.CPBaddr` fields must be set.

```
typedef struct s_pxe_cpb_station_address {

  // If supplied and supported, the current station MAC address
  // will be changed.

  PXE_MAC_ADDR   StationAddr;
} PXE_CPB_STATION_ADDRESS;
```

E.4.12.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. DB is written.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.12.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully.
INVALID_CDB	One of the CDB fields was not set correctly.
INVALID_CPB	One of the CPB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.
UNSUPPORTED	The requested operation is not supported.

Before Using the DB

The DB is used to read the current station, broadcast and permanent station MAC addresses. The `CDB.DBsize` and `CDB.DBaddr` fields can be set to `PXE_DBSIZE_NOT_USED` and `PXE_DBADDR_NOT_USED` if these addresses do not need to be read.

```
typedef struct s_pxe_db_station_address {

  // Current station MAC address.
  PXE_MAC_ADDR   StationAddr;

  // Station broadcast MAC address.
  PXE_MAC_ADDR   BroadcastAddr;

  // Permanent station MAC address.
  PXE_MAC_ADDR   PermanentAddr;
} PXE_DB_STATION_ADDRESS;
```

E.4.13 Statistics

This command is used to read and clear the NIC traffic statistics. Before using this command check to see if statistics is supported in the **!PXE.Implementation** flags.

E.4.13.1 Issuing the Command

To issue a Statistics command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a Statistics command
OpCode	**PXE_OPCODE_STATISTICS**
OpFlags	Set as needed.
CPBsize	**PXE_CPBSIZE_NOT_USED**
DBsize	**sizeof(PXE_DB_STATISTICS)**
CPBaddr	**PXE_CPBADDR_NOT_USED**
DBaddr	Address of **PXE_DB_STATISTICS** structure.
StatCode	**PXE_STATCODE_INITIALIZE**
StatFlags	**PXE_STATFLAGS_INITIALIZE**
IFnum	A valid interface number from zero to **!PXE.IFcnt**.
Control	Set as needed.

OpFlags

To read the current statistics counters set the OpFlags field to:

PXE_OPFLAGS_STATISTICS_READ

To reset the current statistics counters set the OpFlags field to:

PXE_OPFLAGS_STATISTICS_RESET

E.4.13.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the **CDB.StatFlags** field. Until these bits change to report **PXE_STATFLAGS_COMMAND_COMPLETE** or **PXE_STATFLAGS_COMMAND_FAILED**, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. DB is written.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.13.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. DB is written.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.
UNSUPPORTED	This command is not supported.

DB

Unsupported statistics counters will be zero filled by UNDI.

```
typedef struct s_pxe_db_statistics {

  // Bit field identifying what statistic data is collected by
  // the UNDI/NIC.
  // If bit 0x00 is set, Data[0x00] is collected.
  // If bit 0x01 is set, Data[0x01] is collected.
  // If bit 0x20 is set, Data[0x20] is collected.
  // If bit 0x21 is set, Data[0x21] is collected.
  // Etc.
  PXE_UINT64     Supported;

  // Statistic data.

  PXE_UINT64     Data[64];
} PXE_DB_STATISTICS;

// Total number of frames received.  Includes frames with errors
// and dropped frames.
#define PXE_STATISTICS_RX_TOTAL_FRAMES                  0x00

// Number of valid frames received and copied into receive
// buffers.
#define PXE_STATISTICS_RX_GOOD_FRAMES                   0x01

// Number of frames below the minimum length for the media.
// This would be <64 for ethernet.
#define PXE_STATISTICS_RX_UNDERSIZE_FRAMES              0x02
```

```
// Number of frames longer than the maxminum length for the
// media.  This would be >1500 for ethernet.
#define PXE_STATISTICS_RX_OVERSIZE_FRAMES              0x03

// Valid frames that were dropped because receive buffers
// were full.
#define PXE_STATISTICS_RX_DROPPED_FRAMES               0x04

// Number of valid unicast frames received and not dropped.
#define PXE_STATISTICS_RX_UNICAST_FRAMES               0x05

// Number of valid broadcast frames received and not dropped.
#define PXE_STATISTICS_RX_BROADCAST_FRAMES             0x06

// Number of valid mutlicast frames received and not dropped.
#define PXE_STATISTICS_RX_MULTICAST_FRAMES             0x07

// Number of frames w/ CRC or alignment errors.
#define PXE_STATISTICS_RX_CRC_ERROR_FRAMES             0x08

// Total number of bytes received.  Includes frames with errors
// and dropped frames.
#define PXE_STATISTICS_RX_TOTAL_BYTES                  0x09

// Transmit statistics.
#define PXE_STATISTICS_TX_TOTAL_FRAMES                 0x0A
#define PXE_STATISTICS_TX_GOOD_FRAMES                  0x0B
#define PXE_STATISTICS_TX_UNDERSIZE_FRAMES             0x0C
#define PXE_STATISTICS_TX_OVERSIZE_FRAMES              0x0D
#define PXE_STATISTICS_TX_DROPPED_FRAMES               0x0E
#define PXE_STATISTICS_TX_UNICAST_FRAMES               0x0F
#define PXE_STATISTICS_TX_BROADCAST_FRAMES             0x10
#define PXE_STATISTICS_TX_MULTICAST_FRAMES             0x11
#define PXE_STATISTICS_TX_CRC_ERROR_FRAMES             0x12
#define PXE_STATISTICS_TX_TOTAL_BYTES                  0x13

// Number of collisions detection on this subnet.
#define PXE_STATISTICS_COLLISIONS                      0x14

// Number of frames destined for unsupported protocol.
#define PXE_STATISTICS_UNSUPPORTED_PROTOCOL            0x15
```

E.4.14 MCast IP To MAC

Translate a multicast IPv4 or IPv6 address to a multicast MAC address.

E.4.14.1 Issuing the Command

To issue a MCast IP To MAC command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a MCast IP To MAC command
OpCode	`PXE_OPCODE_MCAST_IP_TO_MAC`
OpFlags	Set as needed.
CPBsize	`sizeof(PXE_CPB_MCAST_IP_TO_MAC)`
DBsize	`sizeof(PXE_DB_MCAST_IP_TO_MAC)`
CPBaddr	Address of `PXE_CPB_MCAST_IP_TO_MAC` structure.
Dbaddr	Address of `PXE_DB_MCAST_IP_TO_MAC` structure.
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
Ifnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

OpFlags

To convert a multicast IP address to a multicast MAC address the UNDI needs to know the format of the IP address. Set one of these OpFlags to identify the format of the IP addresses in the CPB:

```
PXE_OPFLAGS_MCAST_IPV4_TO_MAC
PXE_OPFLAGS_MCAST_IPV6_TO_MAC
```

Preparing the CPB

Fill in an array of one or more multicast IP addresses. Be sure to set the `CDB.CPBsize` and `CDB.CPBaddr` fields accordingly.

```
typedef struct s_pxe_cpb_mcast_ip_to_mac {

   // Multicast IP address to be converted to multicast
   // MAC address.
   PXE_IP_ADDR    IP[n];
} PXE_CPB_MCAST_IP_TO_MAC;
```

E.4.14.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. DB is written.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.14.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. DB is written.
INVALID_CDB	One of the CDB fields was not set correctly.
INVALID_CPB	One of the CPB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.

Before Using the DB

The DB is where the multicast MAC addresses will be written.

```
typedef struct s_pxe_db_mcast_ip_to_mac {

  // Multicast MAC address.

  PXE_MAC_ADDR   MAC[n];
} PXE_DB_MCAST_IP_TO_MAC;
```

E.4.15 NvData

This command is used to read and write (if supported by NIC H/W) nonvolatile storage on the NIC. Nonvolatile storage could be EEPROM, FLASH or battery backed RAM.

E.4.15.1 Issuing the Command

To issue a NvData command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a NvData command
OpCode	`PXE_OPCODE_NVDATA`
OpFlags	Set as needed.
CPBsize	`sizeof(PXE_CPB_NVDATA)`
DBsize	`sizeof(PXE_DB_NVDATA)`
CPBaddr	Address of `PXE_CPB_NVDATA` structure.
Dbaddr	Address of `PXE_DB_NVDATA` structure.
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
Ifnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

Preparing the CPB

There are two types of nonvolatile data CPBs, one for sparse updates and one for bulk updates. Sparse updates allow updating of single nonvolatile storage items. Bulk updates always update all nonvolatile storage items. Check the `!PXE.Implementation` flags to see which type of nonvolatile update is supported by this UNDI and network device.

If you do not need to update the nonvolatile storage set the `CDB.CPBsize` and `CDB.CPBaddr` fields to `PXE_CPBSIZE_NOT_USED` and `PXE_CPBADDR_NOT_USED`.

Sparse NvData CPB

```
typedef struct s_pxe_cpb_nvdata_sparse {
  // NvData item list.  Only items in this list will be updated.

  struct {

     // Nonvolatile storage address to be changed.
     PXE_UINT32       Addr;

     // Data item to write into above storage address.
     union {
       PXE_UINT8      Byte;
       PXE_UINT16     Word;
       PXE_UINT32     Dword;
     } Data;
  } Item[n];
} PXE_CPB_NVDATA_SPARSE;
```

Bulk NvData CPB

```
// When using bulk update, the size of the CPB structure must be
// the same size as the nonvolatile NIC storage.

typedef union u_pxe_cpb_nvdata_bulk {

  // Array of byte-wide data items.
  PXE_UINT8      Byte[n];

  // Array of word-wide data items.
  PXE_UINT16     Word[n];

  // Array of dword-wide data items.
  PXE_UINT32     Dword[n];
} PXE_CPB_NVDATA_BULK;
```

E.4.15.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. Nonvolatile data is updated from CPB and/or written to DB.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.15.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. Nonvolatile data is updated from CPB and/or written to DB.
INVALID_CDB	One of the CDB fields was not set correctly.
INVALID_CPB	One of the CPB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.
UNSUPPORTED	Requested operation is unsupported.

DB

Check the width and number of nonvolatile storage items. This information is returned by the Get Init Info command.

```
typedef struct s_pxe_db_nvdata {

  // Arrays of data items from nonvolatile storage.
  union {

      // Array of byte-wide data items.
      PXE_UINT8  Byte[n];

      // Array of word-wide data items.
      PXE_UINT16 Word[n];

      // Array of dword-wide data items.
      PXE_UINT32 Dword[n];
  } Data;
} PXE_DB_NVDATA;
```

E.4.16 Get Status

This command returns the current interrupt status and/or the transmitted buffer addresses. If the current interrupt status is returned, pending interrupts will be acknowledged by this command. Transmitted buffer addresses that are written to the DB are removed from the transmitted buffer queue.

This command may be used in a polled fashion with external interrupts disabled.

E.4.16.1 Issuing the Command

To issue a Get Status command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a Get Status command
OpCode	`PXE_OPCODE_GET_STATUS`
OpFlags	Set as needed.
CPBsize	`PXE_CPBSIZE_NOT_USED`
DBsize	`Sizeof(PXE_DB_GET_STATUS)`
CPBaddr	`PXE_CPBADDR_NOT_USED`
DBaddr	Address of `PXE_DB_GET_STATUS` structure.
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

Setting OpFlags

Set one or both of the OpFlags below to return the interrupt status and/or the transmitted buffer addresses.

```
PXE_OPFLAGS_GET_INTERRUPT_STATUS
PXE_OPFLAGS_GET_TRANSMITTED_BUFFERS
```

E.4.16.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. StatFlags and/or DB are updated.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.16.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. StatFlags and/or DB are updated.
INVALID_CDB	One of the CDB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.

StatFlags

If the command completes successfully and the `PXE_OPFLAGS_GET_INTERRUPT_STATUS` OpFlag was set in the CDB, the current interrupt status is returned in the `CDB.StatFlags` field and any pending interrupts will have been cleared.

```
PXE_STATFLAGS_GET_STATUS_RECEIVE
PXE_STATFLAGS_GET_STATUS_TRANSMIT
PXE_STATFLAGS_GET_STATUS_COMMAND
PXE_STATFLAGS_GET_STATUS_SOFTWARE
```

The StatFlags above may not map directly to external interrupt signals. For example: Some NICs may combine both the receive and transmit interrupts to one external interrupt line. When a receive and/or transmit interrupt occurs, use the Get Status to determine which type(s) of interrupt(s) occurred.

This flag is set if the transmitted buffer queue is empty. This flag will be set if all transmitted buffer addresses get written t into the DB.

```
PXE_STATFLAGS_GET_STATUS_TXBUF_QUEUE_EMPTY
```

This flag is set if no transmitted buffer addresses were written into the DB.

```
PXE_STATFLAGS_GET_STATUS_NO_TXBUFS_WRITTEN
```

Using the DB

When reading the transmitted buffer addresses there should be room for at least one 64-bit address in the DB. Once a complete transmitted buffer address is written into the DB, the address is removed from the transmitted buffer queue. If the transmitted buffer queue is full, attempts to use the Transmit command will fail.

```
#pragma pack(1)
typedef struct s_pxe_db_get_status {

  // Length of next receive frame (header + data).  If this is
  // zero, there is no next receive frame available.

  PXE_UINT32     RxFrameLen;

  // Reserved, set to zero.

  PXE_UINT32     reserved;

  // Addresses of transmitted buffers that need to be recycled.

  PXE_UINT64     xBuffer[n];
} PXE_DB_GET_STATUS;
#pragma pack()
```

E.4.17 Fill Header

This command is used to fill the media header(s) in transmit packet(s).

E.4.17.1 Issuing the Command

To issue a Fill Header command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a Fill Header command
OpCode	`PXE_OPCODE_FILL_HEADER`
OpFlags	Set as needed.
CPBsize	`PXE_CPB_FILL_HEADER`
DBsize	`PXE_DBSIZE_NOT_USED`
CPBaddr	Address of a `PXE_CPB_FILL_HEADER` structure.
DBaddr	`PXE_DBADDR_NOT_USED`
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

OpFlags

Select one of the OpFlags below so the UNDI knows what type of CPB is being used.

```
PXE_OPFLAGS_FILL_HEADER_WHOLE
PXE_OPFLAGS_FILL_HEADER_FRAGMENTED
```

Preparing the CPB

If multiple frames per command are supported (see `!PXE.Implementation` flags), multiple CPBs can be packed together. The `CDB.CPBsize` field lets the UNDI know how many CPBs are packed together.

Nonfragmented Frame

```
#pragma pack(1)
typedef struct s_pxe_cpb_fill_header {

   // Source and destination MAC addresses.  These will be copied
   // into the media header without doing byte swapping.
   PXE_MAC_ADDR   SrcAddr;
   PXE_MAC_ADDR   DestAddr;

   // Address of first byte of media header.  The first byte of
   // packet data follows the last byte of the media header.
   PXE_UINT64     MediaHeader;
```

```
    // Length of packet data in bytes (not including the media
    // header).
    PXE_UINT32     PacketLen;

    // Protocol type.  This will be copied into the media header
    // without doing byte swapping.  Protocol type numbers can be
    // obtained from the Assigned Numbers RFC 1700.
    PXE_UINT16     Protocol;

    // Length of the media header in bytes.
    PXE_UINT16     MediaHeaderLen;
  } PXE_CPB_FILL_HEADER;
  #pragma pack()

  #define PXE_PROTOCOL_ETHERNET_IP                 0x0800
  #define PXE_PROTOCOL_ETHERNET_ARP                0x0806
```

Fragmented Frame

```
  #pragma pack(1)
  typedef struct s_pxe_cpb_fill_header_fragmented {

    // Source and destination MAC addresses.  These will be copied
    // into the media header without doing byte swapping.
    PXE_MAC_ADDR   SrcAddr;
    PXE_MAC_ADDR   DestAddr;

    // Length of packet data in bytes (not including the media
    // header).

    PXE_UINT32     PacketLen;
    // Protocol type.  This will be copied into the media header
    // without doing byte swapping.  Protocol type numbers can be
    // obtained from the Assigned Numbers RFC 1700.
    PXE_MEDIA_PROTOCOL Protocol;

    // Length of the media header in bytes.
    PXE_UINT16     MediaHeaderLen;

    // Number of packet fragment descriptors.
    PXE_UINT16     FragCnt;

    // Reserved, must be set to zero.
    PXE_UINT16     reserved;
    // Array of packet fragment descriptors.  The first byte of the
    // media header is the first byte of the first fragment.

    struct {

      // Address of this packet fragment.
      PXE_UINT64      FragAddr;

      // Length of this packet fragment.
      PXE_UINT32      FragLen;

      // Reserved, must be set to zero.
      PXE_UINT32      reserved;
    } FragDesc[n];
  } PXE_CPB_FILL_HEADER_FRAGMENTED;
  #pragma pack()
```

E.4.17.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. Frame is ready to transmit.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.17.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. Frame is ready to transmit.
INVALID_CDB	One of the CDB fields was not set correctly.
INVALID_CPB	One of the CPB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Try again later.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.

E.4.18 Transmit

The Transmit command is used to place a packet into the transmit queue. The data buffers given to this command are to be considered locked and the application or universal network driver loses the ownership of those buffers and must not free or relocate them until the ownership returns.

When the packets are transmitted, a transmit complete interrupt is generated (if interrupts are disabled, the transmit interrupt status is still set and can be checked using the Get Status command).

Some UNDI implementations and network adapters support transmitting multiple packets with one transmit command. If this feature is supported, multiple transmit CPBs can be linked in one transmit command.

Though all UNDIs support fragmented frames, the same cannot be said for all network devices or protocols. If a fragmented frame CPB is given to UNDI and the network device does not support fragmented frames (see `!PXE.Implementation` flags), the UNDI will have to copy the fragments into a local buffer before transmitting.

E.4.18.1 Issuing the Command

To issue a Transmit command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a Transmit command
OpCode	`PXE_OPCODE_TRANSMIT`
OpFlags	Set as needed.
CPBsize	`sizeof(PXE_CPB_TRANSMIT)`
DBsize	`PXE_DBSIZE_NOT_USED`
CPBaddr	Address of a `PXE_CPB_TRANSMIT` structure.
DBaddr	`PXE_DBADDR_NOT_USED`
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

OpFlags

Check the **!PXE.Implementation** flags to see if the network device support fragmented packets. Select one of the OpFlags below so the UNDI knows what type of CPB is being used.

```
PXE_OPFLAGS_TRANSMIT_WHOLE
PXE_OPFLAGS_TRANSMIT_FRAGMENTED
```

In addition to selecting whether or not fragmented packets are being given, S/W UNDI needs to know if it should block until the packets are transmitted. H/W UNDI cannot block, these two OpFlag settings have no affect when used with H/W UNDI.

```
PXE_OPFLAGS_TRANSMIT_BLOCK
PXE_OPFLAGS_TRANSMIT_DONT_BLOCK
```

Preparing the CPB

If multiple frames per command are supported (see **!PXE.Implementation** flags), multiple CPBs can be packed together. The **CDB.CPBsize** field lets the UNDI know how may frames are to be transmitted.

Nonfragmented Frame

```
#pragma pack(1)
typedef struct s_pxe_cpb_transmit {

  // Address of first byte of frame buffer.  This is also the
  // first byte of the media header.  This address must be a
  // processor-based address for S/W UNDI and a device-based
  // address for H/W UNDI.
  PXE_UINT64     FrameAddr;

  // Length of the data portion of the frame buffer in bytes.  Do
  // not include the length of the media header.
  PXE_UINT32     DataLen;

  // Length of the media header in bytes.
  PXE_UINT16     MediaheaderLen;

  // Reserved, must be zero.
  PXE_UINT16     reserved;
} PXE_CPB_TRANSMIT;
#pragma pack()
```

Fragmented Frame

```
#pragma pack(1)
typedef struct s_pxe_cpb_transmit_fragments {

  // Length of packet data in bytes (not including the media
  // header).
  PXE_UINT32     FrameLen;

  // Length of the media header in bytes.
  PXE_UINT16     MediaheaderLen;

  // Number of packet fragment descriptors.
  PXE_UINT16     FragCnt;

  // Array of frame fragment descriptors.  The first byte of the
  // first fragment is also the first byte of the media header.
  struct {
     // Address of this frame fragment.  This address must be a
     // processor-based address for S/W UNDI and a device-based
     // address for H/W UNDI.
     PXE_UINT64      FragAddr;

     // Length of this frame fragment.
     PXE_UINT32      FragLen;

     // Reserved, must be set to zero.
     PXE_UINT32      reserved;
  } FragDesc[n];
} PXE_CPB_TRANSMIT_FRAGMENTS;
#pragma pack()
```

E.4.18.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the **CDB.StatFlags** field. Until these bits change to report **PXE_STATFLAGS_COMMAND_COMPLETE** or **PXE_STATFLAGS_COMMAND_FAILED**, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. Use the Get Status command to see when frame buffers can be reused.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.18.3 Checking Command Execution Results

After command execution completes, either successfully or not, the **CDB.StatCode** field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. Use the Get Status command to see when frame buffers can be reused.
INVALID_CDB	One of the CDB fields was not set correctly.
INVALID_CPB	One of the CPB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Wait for queued commands to complete. Try again later.
BUFFER_FULL	Transmit buffer is full. Call Get Status command to empty buffer.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.

E.4.19 Receive

When the network adapter has received a frame, this command is used to copy the frame into driver/application storage. Once a frame has been copied, it is removed from the receive queue.

E.4.19.1 Issuing the Command

To issue a Receive command, create a CDB and fill it in as shown in the table below:

CDB Field	How to initialize the CDB structure for a Receive command
OpCode	`PXE_OPCODE_RECEIVE`
OpFlags	Set as needed.
CPBsize	`sizeof(PXE_CPB_RECEIVE)`
DBsize	`sizeof(PXE_DB_RECEIVE)`
CPBaddr	Address of a `PXE_CPB_RECEIVE` structure.
DBaddr	Address of a `PXE_DB_RECEIVE` structure.
StatCode	`PXE_STATCODE_INITIALIZE`
StatFlags	`PXE_STATFLAGS_INITIALIZE`
IFnum	A valid interface number from zero to `!PXE.IFcnt`.
Control	Set as needed.

Preparing the CPB

If multiple frames per command are supported (see `!PXE.Implementation` flags), multiple CPBs can be packed together. For each complete received frame, a receive buffer large enough to contain the entire unfragmented frame needs to be described in the CPB. Note that if a smaller than required buffer is provided, only a portion of the packet is received into the buffer, and the remainder of the packet is lost. Subsequent attempts to receive the same packet with a corrected (larger) buffer will be unsuccessful, because the packet will have been flushed from the queue.

```
#pragma pack(1)
typedef struct s_pxe_cpb_receive {

  // Address of first byte of receive buffer.  This is also the
  // first byte of the frame header.  This address must be a
  // processor-based address for S/W UNDI and a device-based
  // address for H/W UNDI.

  PXE_UINT64     BufferAddr;

  // Length of receive buffer.  This must be large enough to hold
  // the received frame (media header + data).  If the length of
  // smaller than the received frame, data will be lost.
  PXE_UINT32     BufferLen;

  // Reserved, must be set to zero.
  PXE_UINT32     reserved;
} PXE_CPB_RECEIVE;
#pragma pack()
```

E.4.19.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the **CDB.StatFlags** field. Until these bits change to report **PXE_STATFLAGS_COMMAND_COMPLETE** or **PXE_STATFLAGS_COMMAND_FAILED**, the command has not been executed by the UNDI.

StatFlags	Reason
COMMAND_COMPLETE	Command completed successfully. Frames received and DB is written.
COMMAND_FAILED	Command failed. StatCode field contains error code.
COMMAND_QUEUED	Command has been queued.
INITIALIZE	Command has been not executed or queued.

E.4.19.3 Checking Command Execution Results

After command execution completes, either successfully or not, the **CDB.StatCode** field contains the result of the command execution.

StatCode	Reason
SUCCESS	Command completed successfully. Frames received and DB is written.
INVALID_CDB	One of the CDB fields was not set correctly.
INVALID_CPB	One of the CPB fields was not set correctly.
BUSY	UNDI is already processing commands. Try again later.
QUEUE_FULL	Command queue is full. Wait for queued commands to complete. Try again later.
NO_DATA	Receive buffers are empty.
NOT_STARTED	The UNDI is not started.
NOT_INITIALIZED	The UNDI is not initialized.

Using the DB

If multiple frames per command are supported (see **!PXE.Implementation** flags), multiple DBs can be packed together.

```
#pragma pack(1)
typedef struct s_pxe_db_receive {

  // Source and destination MAC addresses from media header.
  PXE_MAC_ADDR   SrcAddr;
  PXE_MAC_ADDR   DestAddr;

  // Length of received frame.  May be larger than receive buffer
  // size.  The receive buffer will not be overwritten.  This is
  // how to tell if data was lost because the receive buffer was
  // too small.
  PXE_UINT32     FrameLen;

  // Protocol type from media header.
  PXE_PROTOCOL   Protocol;

  // Length of media header in received frame.
  PXE_UINT16     MediaHeaderLen;

  // Type of receive frame.
  PXE_FRAME_TYPE Type;

  // Reserved, must be zero.
  PXE_UINT8      reserved[7];
} PXE_DB_RECEIVE;
#pragma pack()
```

E.5 UNDI as an EFI Runtime Driver

This section defines the interface between UNDI and EFI and how UNDI must be initialized as an EFI runtime driver.

In the EFI environment, UNDI must implement the Network Interface Identifier (NII) protocol and install an interface pointer of the type NII protocol with EFI. It must also install a device path protocol with a device path that includes the hardware device path (such as PCI) appended with the NIC's MAC address. If the UNDI drives more than one NIC device, it must install one set of NII and device path protocols for each device it controls.

UNDI must be compiled as a runtime driver so that when the operating system loads, a universal protocol driver can use the UNDI driver to access the NIC hardware.

For the universal driver to be able to find UNDI, UNDI must install a configuration table (using the EFI boot service **InstallConfigurationTable()**) for the GUID **NETWORK_INTERFACE_IDENTIFIER_PROTOCOL**. The format of the configuration table for UNDI is defined as follows.

```
struct  undiconfig_table {
      UINT32 NumberOfInterfaces; // The number of NIC devices
                                 // that this UNDI controls.
      UINT32 reserved;
      struct undiconfigtable *nextlink;
                        // A pointer to the next UNDI
                        // configuration table.
      struct {
            VOID *NII_InterfacePointer;
                        // Pointer to the NII interface structure.
            VOID *DevicePathPointer;
                        // pointer to the device path for this NIC
      } NII_entry[n];   // The length of this array is given in
                        // the NumberOfInterfaces field.
} UNDI_CONFIG_TABLE;
```

Since there can only be one configuration table associated with any GUID and there can be more than one UNDI loaded, every instance of UNDI must check for any previous installations of the configuration tables and if there are any, it must traverse through the list of all UNDI configuration tables using the nextlink and install itself as the nextlink of the last table in the list.

The universal protocol driver is responsible for converting all the pointers in the UNDI_CONFIGURATION_TABLE to virtual addresses before accessing them. However, UNDI must install an event handler for the SET_VIRTUAL_ADDRESS event and convert all its internal pointers into virtual addresses when the event occurs for the universal protocol driver to be able to use UNDI.

Appendix F
Using the Simple Pointer Protocol

The Simple Pointer Protocol is intended to provide a simple mechanism for an EFI application to interact with the user with some type of pointer device. To keep this interface simple, many of the custom controls that are typically present in an OS-present environment were left out. This includes the ability to adjust the double-click speed and the ability to adjust the pointer speed. Instead, the recommendations for how the Simple Pointer Protocol should be used are listed here.

X-Axis Movement:

If the Simple Pointer Protocol is being used to move a pointer or cursor around on an output display, the movement along the x-axis should move the pointer or cursor horizontally.

Y-Axis Movement:

If the Simple Pointer Protocol is being used to move a pointer or cursor around on an output display, the movement along the y-axis should move the pointer or cursor vertically.

Z-Axis Movement:

If the Simple Pointer Protocol is being used to move a pointer or cursor around on an output display, and the application that is using the Simple Pointer Protocol supports scrolling, then the movement along the z-axis should scroll the output display.

Double Click Speed:

If two clicks of the same button on a pointer occur in less than 0.5 seconds, then a double-click event has occurred. If a the same button is pressed with more than 0.5 seconds between clicks, then this is interpreted as two single-click events.

Pointer Speed:

The Simple Pointer Protocol returns the movement of the pointer device along an axis in counts. The Simple Pointer Protocol also contains a set of resolution fields that define the number of counts that will be received for each millimeter of movement of the pointer device along an axis. From these two values, the consumer of this protocol can determine the distance the pointer device has been moved in millimeters along an axis. For most applications, movement of a pointer device will result in the movement of a pointer on the screen. For each millimeter of motion by the pointer device in the x-axis, the pointer on the screen will be moved 2 percent of the screen width. For each millimeter of motion by the pointer device in the y-axis, the pointer on the screen will be moved 2 percent of the screen height.

Appendix G
Using the EFI SCSI Pass Thru Protocol

This appendix describes how an EFI utility might gain access to the EFI SCSI Pass Thru interfaces. The basic concept is to use the **LocateHandle()** boot service to retrieve the list of handles that support the **EFI_SCSI_PASS_THRU_Protocol**. Each of these handles represents a different SCSI channel present in the system. Each of these handles can then be used the retrieve the **EFI_SCSI_PASS_THRU_Protocol** interface with the **HandleProtocol()** boot service. The **EFI_SCSI_PASS_THRU_Protocol** interface provides the services required to access any of the SCSI devices attached to a SCSI channel. The services of the **EFI_SCSI_PASS_THRU_Protocol** are then to loop through the Target IDs of all the SCSI devices on the SCSI channel.

```
#include “efi.h”
#include “efilib.h”

#include EFI_PROTOCOL_DEFINITION(ScsiPassThru)

EFI_GUID gEfiScsiPassThruProtocolGuid = EFI_SCSI_PASS_THRU_PROTOCOL_GUID;

EFI_STATUS
UtilityEntryPoint(
  EFI_HANDLE                ImageHandle,
  EFI_SYSTEM_TABLE          SystemTable
  )
{
  EFI_STATUS                    Status;
  UINTN                         NoHandles;
  EFI_HANDLE                    *HandleBuffer;
  UINTN                         Index;
  EFI_SCSI_PASS_THRU_PROTOCOL   *ScsiPassThruProtocol;

  //
  // Initialize EFI Library
  //
  InitializeLib (ImageHandle, SystemTable);

  //
  // Get list of handles that support the
  // EFI_SCSI_PASS_THRU_PROTOCOL
  //
  NoHandles = 0;
  HandleBuffer = NULL;
  Status = LibLocateHandle(
             ByProtocol,
             &gEfiScsiPassThruProtocolGuid,
             NULL,
             &NoHandles,
             &HandleBuffer
             );
  if (EFI_ERROR(Status)) {
    BS->Exit(ImageHandle,EFI_SUCCESS,0,NULL);
  }

  //
  // Loop through all the handles that support
  // EFI_SCSI_PASS_THRU
  //
  for (Index = 0; Index < NoHandles; Index++) {

    //
    // Get the EFI_SCSI_PASS_THRU_PROTOCOL Interface
    // on each handle
    //
    BS->HandleProtocol(
          HandleBuffer[Index],
          &gEfiScsiPassThruProtocolGuid,
          (VOID **)&ScsiPassThruProtocol
          );

    if (!EFI_ERROR(Status)) {

      //
      // Use the EFI_SCSI_PASS_THRU  Interface to
      // perform tests
      //
      Status = DoScsiTests(ScsiPassThruProtocol);
    }
  }
  return EFI_SUCCESS;
}

EFI_STATUS
DoScsiTests(
  EFI_SCSI_PASS_THRU _PROTOCOL  *ScsiPassThruProtocol
  )

{
  EFI_STATUS                        Status;
  UINT32                            Target;
  UINT64                            Lun;
  EFI_SCSI_PASS_THRU_SCSI_REQUEST_PACKET  Packet;
  EFI_EVENT                         Event;
  //
  // Get first Target ID and LUN on the SCSI channel
  //
  Target = 0xffffffff;
  Lun    = 0;
  Status = ScsiPassThruProtocol->GetNextDevice(
                                   ScsiPassThruProtocol,
                                   &Target,
                                   &Lun
                                   );

  //
  // Loop through all the SCSI devices on the SCSI channel
  //
  while (!EFI_ERROR (Status)) {

    //
    // Blocking I/O example.
    // Fill in Packet before calling PassThru()
    //
    Status = ScsiPassThruProtocol->PassThru(
                                   ScsiPassThruProtocol,
                                   Target,
                                   Lun,
                                   &Packet,
                                   NULL
                                   );

    //
    // Non Blocking I/O
    // Fill in Packet and create Event before calling PassThru()
    //
    Status = ScsiPassThruProtocol->PassThru(
                                   ScsiPassThruProtocol,
                                   Target,
                                   Lun,
                                   &Packet,
                                   &Event
                                   );

    //
    // Get next Target ID and LUN on the SCSI channel
    //
    Status = ScsiPassThruProtocol->GetNextDevice(
                                   ScsiPassThruProtocol,
                                   &Target,
                                   &Lun
                                   );
  }

  return EFI_SUCCESS;
}
```

Appendix H
Compression Source Code

```
/*++

Copyright (c) 2001-2002  Intel Corporation

Module Name:

  Compress.c

Abstract:

  Compression routine.  The compression algorithm is a mixture of
  LZ77 and Huffman Coding.  LZ77 transforms the source data into a
  sequence of Original Characters and Pointers to repeated strings.
  This sequence is further divided into Blocks and Huffman codings
  are applied to each Block.

Revision History:
--*/

#include <string.h>
#include <stdlib.h>
#include "eficommon.h"


//
// Macro Definitions
//

typedef INT16             NODE;
#define UINT8_MAX         0xff
#define UINT8_BIT         8
#define THRESHOLD         3
#define INIT_CRC          0
#define WNDBIT            13
#define WNDSIZ            (1U << WNDBIT)
#define MAXMATCH          256
#define PERC_FLAG         0x8000U
#define CODE_BIT          16
#define NIL               0
#define MAX_HASH_VAL      (3 * WNDSIZ + (WNDSIZ / 512 + 1) * UINT8_MAX)
#define HASH(p, c)        ((p) + ((c) << (WNDBIT - 9)) + WNDSIZ * 2)
#define CRCPOLY           0xA001
#define UPDATE_CRC(c)     mCrc = mCrcTable[(mCrc ^ (c)) & 0xFF] ^ (mCrc >>
UINT8_BIT)

//
// C: the Char&Len Set; P: the Position Set; T: the exTra Set
//

#define NC                (UINT8_MAX + MAXMATCH + 2 - THRESHOLD)
#define CBIT              9
#define NP                (WNDBIT + 1)
#define PBIT              4
```

```
#define NT                  (CODE_BIT + 3)
#define TBIT                5
#if NT > NP
  #define                   NPT NT
#else
  #define                   NPT NP
#endif

//
// Function Prototypes
//

STATIC
VOID
PutDword(
  IN UINT32 Data
  );

STATIC
EFI_STATUS
AllocateMemory (
  );

STATIC
VOID
FreeMemory (
  );

STATIC
VOID
InitSlide (
  );

STATIC
NODE
Child (
  IN NODE q,
  IN UINT8 c
  );

STATIC
VOID
MakeChild (
  IN NODE q,
  IN UINT8 c,
  IN NODE r
  );

STATIC
VOID
Split (
  IN NODE Old
  );

STATIC
VOID
InsertNode (
  );
```

```
STATIC
VOID
DeleteNode (
  );

STATIC
VOID
GetNextMatch (
  );

STATIC
EFI_STATUS
Encode (
  );

STATIC
VOID
CountTFreq (
  );

STATIC
VOID
WritePTLen (
  IN INT32 n,
  IN INT32 nbit,
  IN INT32 Special
  );

STATIC
VOID
WriteCLen (
  );

STATIC
VOID
EncodeC (
  IN INT32 c
  );

STATIC
VOID
EncodeP (
  IN UINT32 p
  );

STATIC
VOID
SendBlock (
  );

STATIC
VOID
Output (
  IN UINT32 c,
  IN UINT32 p
  );
```

```
STATIC
VOID
HufEncodeStart (
  );

STATIC
VOID
HufEncodeEnd (
  );

STATIC
VOID
MakeCrcTable (
  );

STATIC
VOID
PutBits (
  IN INT32 n,
  IN UINT32 x
  );

STATIC
INT32
FreadCrc (
  OUT UINT8 *p,
  IN  INT32 n
  );

STATIC
VOID
InitPutBits (
  );

STATIC
VOID
CountLen (
  IN INT32 i
  );

STATIC
VOID
MakeLen (
  IN INT32 Root
  );

STATIC
VOID
DownHeap (
  IN INT32 i
  );

STATIC
VOID
MakeCode (
  IN  INT32 n,
  IN  UINT8 Len[],
  OUT UINT16 Code[]
  );
```

```
STATIC
INT32
MakeTree (
  IN  INT32   NParm,
  IN  UINT16  FreqParm[],
  OUT UINT8   LenParm[],
  OUT UINT16  CodeParm[]
  );


//
//  Global Variables
//

STATIC UINT8  *mSrc, *mDst, *mSrcUpperLimit, *mDstUpperLimit;

STATIC UINT8  *mLevel, *mText, *mChildCount, *mBuf, mCLen[NC], mPTLen[NPT],
*mLen;
STATIC INT16  mHeap[NC + 1];
STATIC INT32  mRemainder, mMatchLen, mBitCount, mHeapSize, mN;
STATIC UINT32 mBufSiz = 0, mOutputPos, mOutputMask, mSubBitBuf, mCrc;
STATIC UINT32 mCompSize, mOrigSize;

STATIC UINT16 *mFreq, *mSortPtr, mLenCnt[17], mLeft[2 * NC - 1], mRight[2 * NC
- 1],
              mCrcTable[UINT8_MAX + 1], mCFreq[2 * NC - 1], mCTable[4096],
mCCode[NC],
              mPFreq[2 * NP - 1], mPTCode[NPT], mTFreq[2 * NT - 1];

STATIC NODE   mPos, mMatchPos, mAvail, *mPosition, *mParent, *mPrev, *mNext =
NULL;


//
// functions
//

EFI_STATUS
Compress (
  IN      UINT8   *SrcBuffer,
  IN      UINT32  SrcSize,
  IN      UINT8   *DstBuffer,
  IN OUT  UINT32  *DstSize
  )
/*++

Routine Description:

  The main compression routine.

Arguments:

  SrcBuffer   - The buffer storing the source data
  SrcSize     - The size of the source data
  DstBuffer   - The buffer to store the compressed data
  DstSize     - On input, the size of DstBuffer; On output,
                the size of the actual compressed data.
```

```
Returns:

  EFI_BUFFER_TOO_SMALL  - The DstBuffer is too small.  In this case,
                 DstSize contains the size needed.
  EFI_SUCCESS           - Compression is successful.

--*/
{
  EFI_STATUS Status = EFI_SUCCESS;

  //
  // Initializations
  //

  mBufSiz = 0;
  mBuf = NULL;
  mText       = NULL;
  mLevel      = NULL;
  mChildCount = NULL;
  mPosition   = NULL;
  mParent     = NULL;
  mPrev       = NULL;
  mNext       = NULL;


  mSrc = SrcBuffer;
  mSrcUpperLimit = mSrc + SrcSize;
  mDst = DstBuffer;
  mDstUpperLimit = mDst + *DstSize;

  PutDword(0L);
  PutDword(0L);

  MakeCrcTable ();

  mOrigSize = mCompSize = 0;
  mCrc = INIT_CRC;

  //
  // Compress it
  //

  Status = Encode();
  if (EFI_ERROR (Status)) {
    return EFI_OUT_OF_RESOURCES;
  }

  //
  // Null terminate the compressed data
  //
  if (mDst < mDstUpperLimit) {
    *mDst++ = 0;
  }

  //
  // Fill in compressed size and original size
  //
  mDst = DstBuffer;
  PutDword(mCompSize+1);
  PutDword(mOrigSize);
  //
  // Return
  //

  if (mCompSize + 1 + 8 > *DstSize) {
    *DstSize = mCompSize + 1 + 8;
    return EFI_BUFFER_TOO_SMALL;
  } else {
    *DstSize = mCompSize + 1 + 8;
    return EFI_SUCCESS;
  }

}

STATIC
VOID
PutDword(
  IN UINT32 Data
  )
/*++

Routine Description:

  Put a dword to output stream

Arguments:

  Data     - the dword to put

Returns: (VOID)

--*/
{
  if (mDst < mDstUpperLimit) {
    *mDst++ = (UINT8)(((UINT8)(Data         )) & 0xff);
  }

  if (mDst < mDstUpperLimit) {
    *mDst++ = (UINT8)(((UINT8)(Data >> 0x08)) & 0xff);
  }

  if (mDst < mDstUpperLimit) {
    *mDst++ = (UINT8)(((UINT8)(Data >> 0x10)) & 0xff);
  }

  if (mDst < mDstUpperLimit) {
    *mDst++ = (UINT8)(((UINT8)(Data >> 0x18)) & 0xff);
  }
}

STATIC
EFI_STATUS
AllocateMemory ()
/*++
Routine Description:

  Allocate memory spaces for data structures used in compression process

Argements: (VOID)

Returns:

  EFI_SUCCESS           - Memory is allocated successfully
  EFI_OUT_OF_RESOURCES  - Allocation fails

--*/
{
  UINT32      i;

  mText = malloc (WNDSIZ * 2 + MAXMATCH);
  for (i = 0; i < WNDSIZ * 2 + MAXMATCH; i ++) {
    mText[i] = 0;
  }
  mLevel      = malloc ((WNDSIZ + UINT8_MAX + 1) * sizeof(*mLevel));
  mChildCount = malloc ((WNDSIZ + UINT8_MAX + 1) * sizeof(*mChildCount));
  mPosition   = malloc ((WNDSIZ + UINT8_MAX + 1) * sizeof(*mPosition));
  mParent     = malloc (WNDSIZ * 2 * sizeof(*mParent));
  mPrev       = malloc (WNDSIZ * 2 * sizeof(*mPrev));
  mNext       = malloc ((MAX_HASH_VAL + 1) * sizeof(*mNext));

  mBufSiz = 16 * 1024U;
  while ((mBuf = malloc(mBufSiz)) == NULL) {
    mBufSiz = (mBufSiz / 10U) * 9U;
    if (mBufSiz < 4 * 1024U) {
      return EFI_OUT_OF_RESOURCES;
    }
  }
  mBuf[0] = 0;

  return EFI_SUCCESS;
}

VOID
FreeMemory ()
/*++

Routine Description:

  Called when compression is completed to free memory previously allocated.

Arguments: (VOID)

Returns: (VOID)

--*/
{
  if (mText) {
    free (mText);
  }

  if (mLevel) {
    free (mLevel);
  }
  if (mChildCount) {
    free (mChildCount);
  }

  if (mPosition) {
    free (mPosition);
  }

  if (mParent) {
    free (mParent);
  }

  if (mPrev) {
    free (mPrev);
  }

  if (mNext) {
    free (mNext);
  }

  if (mBuf) {
    free (mBuf);
  }

  return;
}


STATIC
VOID
InitSlide ()
/*++

Routine Description:

  Initialize String Info Log data structures

Arguments: (VOID)

Returns: (VOID)

--*/
{
  NODE i;

  for (i = WNDSIZ; i <= WNDSIZ + UINT8_MAX; i++) {
    mLevel[i] = 1;
    mPosition[i] = NIL;  /* sentinel */
  }
  for (i = WNDSIZ; i < WNDSIZ * 2; i++) {
    mParent[i] = NIL;
  }
  mAvail = 1;
  for (i = 1; i < WNDSIZ - 1; i++) {
    mNext[i] = (NODE)(i + 1);
  }
  mNext[WNDSIZ - 1] = NIL;
  for (i = WNDSIZ * 2; i <= MAX_HASH_VAL; i++) {
    mNext[i] = NIL;
  }
}


STATIC
NODE
Child (
  IN NODE q,
  IN UINT8 c
  )
/*++

Routine Description:

  Find child node given the parent node and the edge character

Arguments:

  q       - the parent node
  c       - the edge character

Returns:

  The child node (NIL if not found)

--*/
{
  NODE r;

  r = mNext[HASH(q, c)];
  mParent[NIL] = q;  /* sentinel */
  while (mParent[r] != q) {
    r = mNext[r];
  }

  return r;
}

STATIC
VOID
MakeChild (
  IN NODE q,
  IN UINT8 c,
  IN NODE r
  )
/*++

Routine Description:

  Create a new child for a given parent node.

Arguments:

  q       - the parent node
  c       - the edge character
  r       - the child node
```

```
Returns: (VOID)

--*/
{
  NODE h, t;

  h = (NODE)HASH(q, c);
  t = mNext[h];
  mNext[h] = r;
  mNext[r] = t;
  mPrev[t] = r;
  mPrev[r] = h;
  mParent[r] = q;
  mChildCount[q]++;
}

STATIC
VOID
Split (
  NODE Old
  )
/*++

Routine Description:

  Split a node.

Arguments:

  Old     - the node to split

Returns: (VOID)

--*/
{
  NODE New, t;

  New = mAvail;
  mAvail = mNext[New];
  mChildCount[New] = 0;
  t = mPrev[Old];
  mPrev[New] = t;
  mNext[t] = New;
  t = mNext[Old];
  mNext[New] = t;
  mPrev[t] = New;
  mParent[New] = mParent[Old];
  mLevel[New] = (UINT8)mMatchLen;
  mPosition[New] = mPos;
  MakeChild(New, mText[mMatchPos + mMatchLen], Old);
  MakeChild(New, mText[mPos + mMatchLen], mPos);
}

STATIC
VOID
InsertNode ()
/*++
Routine Description:

  Insert string info for current position into the String Info Log

Arguments: (VOID)

Returns: (VOID)

--*/
{
  NODE q, r, j, t;
  UINT8 c, *t1, *t2;

  if (mMatchLen >= 4) {

    //
    // We have just got a long match, the target tree
    // can be located by MatchPos + 1.  Travese the tree
    // from bottom up to get to a proper starting point.
    // The usage of PERC_FLAG ensures proper node deletion
    // in DeleteNode() later.
    //

    mMatchLen--;
    r = (INT16)((mMatchPos + 1) | WNDSIZ);
    while ((q = mParent[r]) == NIL) {
      r = mNext[r];
    }
    while (mLevel[q] >= mMatchLen) {
      r = q;  q = mParent[q];
    }
    t = q;
    while (mPosition[t] < 0) {
      mPosition[t] = mPos;
      t = mParent[t];
    }
    if (t < WNDSIZ) {
      mPosition[t] = (NODE)(mPos | PERC_FLAG);
    }
  } else {

    //
    // Locate the target tree
    //

    q = (INT16)(mText[mPos] + WNDSIZ);
    c = mText[mPos + 1];
    if ((r = Child(q, c)) == NIL) {
      MakeChild(q, c, mPos);
      mMatchLen = 1;
      return;
    }
    mMatchLen = 2;
  }

  //
  // Traverse down the tree to find a match.
  // Update Position value along the route.
  // Node split or creation is involved.
  //
  for ( ; ; ) {
    if (r >= WNDSIZ) {
      j = MAXMATCH;
      mMatchPos = r;
    } else {
      j = mLevel[r];
      mMatchPos = (NODE)(mPosition[r] & ~PERC_FLAG);
    }
    if (mMatchPos >= mPos) {
      mMatchPos -= WNDSIZ;
    }
    t1 = &mText[mPos + mMatchLen];
    t2 = &mText[mMatchPos + mMatchLen];
    while (mMatchLen < j) {
      if (*t1 != *t2) {
        Split(r);
        return;
      }
      mMatchLen++;
      t1++;
      t2++;
    }
    if (mMatchLen >= MAXMATCH) {
      break;
    }
    mPosition[r] = mPos;
    q = r;
    if ((r = Child(q, *t1)) == NIL) {
      MakeChild(q, *t1, mPos);
      return;
    }
    mMatchLen++;
  }
  t = mPrev[r];
  mPrev[mPos] = t;
  mNext[t] = mPos;
  t = mNext[r];
  mNext[mPos] = t;
  mPrev[t] = mPos;
  mParent[mPos] = q;
  mParent[r] = NIL;

  //
  // Special usage of 'next'
  //
  mNext[r] = mPos;

}

STATIC
VOID
DeleteNode ()
/*++

Routine Description:

  Delete outdated string info.  (The Usage of PERC_FLAG
  ensures a clean deletion)
```

```
Arguments: (VOID)

Returns: (VOID)

--*/
{
  NODE q, r, s, t, u;

  if (mParent[mPos] == NIL) {
    return;
  }

  r = mPrev[mPos];
  s = mNext[mPos];
  mNext[r] = s;
  mPrev[s] = r;
  r = mParent[mPos];
  mParent[mPos] = NIL;
  if (r >= WNDSIZ || --mChildCount[r] > 1) {
    return;
  }
  t = (NODE)(mPosition[r] & ~PERC_FLAG);
  if (t >= mPos) {
    t -= WNDSIZ;
  }
  s = t;
  q = mParent[r];
  while ((u = mPosition[q]) & PERC_FLAG) {
    u &= ~PERC_FLAG;
    if (u >= mPos) {
      u -= WNDSIZ;
    }
    if (u > s) {
      s = u;
    }
    mPosition[q] = (INT16)(s | WNDSIZ);
    q = mParent[q];
  }
  if (q < WNDSIZ) {
    if (u >= mPos) {
      u -= WNDSIZ;
    }
    if (u > s) {
      s = u;
    }
    mPosition[q] = (INT16)(s | WNDSIZ | PERC_FLAG);
  }
  s = Child(r, mText[t + mLevel[r]]);
  t = mPrev[s];
  u = mNext[s];
  mNext[t] = u;
  mPrev[u] = t;
  t = mPrev[r];
  mNext[t] = s;
  mPrev[s] = t;
  t = mNext[r];
  mPrev[t] = s;
  mNext[s] = t;
```

```
    mParent[s] = mParent[r];
    mParent[r] = NIL;
    mNext[r] = mAvail;
    mAvail = r;
  }

  STATIC
  VOID
  GetNextMatch ()
  /*++

  Routine Description:

    Advance the current position (read in new data if needed).
    Delete outdated string info.  Find a match string for current position.

  Arguments: (VOID)

  Returns: (VOID)

  --*/
  {
    INT32 n;

    mRemainder--;
    if (++mPos == WNDSIZ * 2) {
      memmove(&mText[0], &mText[WNDSIZ], WNDSIZ + MAXMATCH);
      n = FreadCrc(&mText[WNDSIZ + MAXMATCH], WNDSIZ);
      mRemainder += n;
      mPos = WNDSIZ;
    }
    DeleteNode();
    InsertNode();
  }

  STATIC
  EFI_STATUS
  Encode ()
  /*++

  Routine Description:

    The main controlling routine for compression process.

  Arguments: (VOID)

  Returns:

    EFI_SUCCESS           - The compression is successful
    EFI_OUT_0F_RESOURCES  - Not enough memory for compression process

  --*/
  {
    EFI_STATUS  Status;
    INT32       LastMatchLen;
    NODE        LastMatchPos;

    Status = AllocateMemory();
    if (EFI_ERROR(Status)) {
      FreeMemory();
      return Status;
    }

    InitSlide();

    HufEncodeStart();

    mRemainder = FreadCrc(&mText[WNDSIZ], WNDSIZ + MAXMATCH);

    mMatchLen = 0;
    mPos = WNDSIZ;
    InsertNode();
    if (mMatchLen > mRemainder) {
      mMatchLen = mRemainder;
    }
    while (mRemainder > 0) {
      LastMatchLen = mMatchLen;
      LastMatchPos = mMatchPos;
      GetNextMatch();
      if (mMatchLen > mRemainder) {
        mMatchLen = mRemainder;
      }

      if (mMatchLen > LastMatchLen || LastMatchLen < THRESHOLD) {

        //
        // Not enough benefits are gained by outputting a pointer,
        // so just output the original character
        //

        Output(mText[mPos - 1], 0);
      } else {

        //
        // Outputting a pointer is beneficial enough, do it.
        //

        Output(LastMatchLen + (UINT8_MAX + 1 - THRESHOLD),
               (mPos - LastMatchPos - 2) & (WNDSIZ - 1));
        while (--LastMatchLen > 0) {
          GetNextMatch();
        }
        if (mMatchLen > mRemainder) {
          mMatchLen = mRemainder;
        }
      }
    }

    HufEncodeEnd();
    FreeMemory();
    return EFI_SUCCESS;
  }

  STATIC
  VOID
  CountTFreq ()
  /*++
  Routine Description:

    Count the frequencies for the Extra Set

  Arguments: (VOID)

  Returns: (VOID)

  --*/
  {
    INT32 i, k, n, Count;

    for (i = 0; i < NT; i++) {
      mTFreq[i] = 0;
    }
    n = NC;
    while (n > 0 && mCLen[n - 1] == 0) {
      n--;
    }
    i = 0;
    while (i < n) {
      k = mCLen[i++];
      if (k == 0) {
        Count = 1;
        while (i < n && mCLen[i] == 0) {
          i++;
          Count++;
        }
        if (Count <= 2) {
          mTFreq[0] = (UINT16)(mTFreq[0] + Count);
        } else if (Count <= 18) {
          mTFreq[1]++;
        } else if (Count == 19) {
          mTFreq[0]++;
          mTFreq[1]++;
        } else {
          mTFreq[2]++;
        }
      } else {
        mTFreq[k + 2]++;
      }
    }
  }

  STATIC
  VOID
  WritePTLen (
    IN INT32 n,
    IN INT32 nbit,
    IN INT32 Special
    )
  /*++

  Routine Description:

    Outputs the code length array for the Extra Set or the Position Set.
```

```
Arguments:

  n       - the number of symbols
  nbit    - the number of bits needed to represent 'n'
  Special - the special symbol that needs to be take care of

Returns: (VOID)

--*/
{
  INT32 i, k;

  while (n > 0 && mPTLen[n - 1] == 0) {
    n--;
  }
  PutBits(nbit, n);
  i = 0;
  while (i < n) {
    k = mPTLen[i++];
    if (k <= 6) {
      PutBits(3, k);
    } else {
      PutBits(k - 3, (1U << (k - 3)) - 2);
    }
    if (i == Special) {
      while (i < 6 && mPTLen[i] == 0) {
        i++;
      }
      PutBits(2, (i - 3) & 3);
    }
  }
}

STATIC
VOID
WriteCLen ()
/*++

Routine Description:

  Outputs the code length array for Char&Length Set

Arguments: (VOID)

Returns: (VOID)

--*/
{
  INT32 i, k, n, Count;

  n = NC;
  while (n > 0 && mCLen[n - 1] == 0) {
    n--;
  }
  PutBits(CBIT, n);
  i = 0;
  while (i < n) {
    k = mCLen[i++];
    if (k == 0) {
      Count = 1;
```

```
          while (i < n && mCLen[i] == 0) {
            i++;
            Count++;
          }
          if (Count <= 2) {
            for (k = 0; k < Count; k++) {
              PutBits(mPTLen[0], mPTCode[0]);
            }
          } else if (Count <= 18) {
            PutBits(mPTLen[1], mPTCode[1]);
            PutBits(4, Count - 3);
          } else if (Count == 19) {
            PutBits(mPTLen[0], mPTCode[0]);
            PutBits(mPTLen[1], mPTCode[1]);
            PutBits(4, 15);
          } else {
            PutBits(mPTLen[2], mPTCode[2]);
            PutBits(CBIT, Count - 20);
          }
        } else {
          PutBits(mPTLen[k + 2], mPTCode[k + 2]);
        }
      }
    }

    STATIC
    VOID
    EncodeC (
      IN INT32 c
      )
    {
      PutBits(mCLen[c], mCCode[c]);
    }

    STATIC
    VOID
    EncodeP (
      IN UINT32 p
      )
    {
      UINT32 c, q;

      c = 0;
      q = p;
      while (q) {
        q >>= 1;
        c++;
      }
      PutBits(mPTLen[c], mPTCode[c]);
      if (c > 1) {
        PutBits(c - 1, p & (0xFFFFU >> (17 - c)));
      }
    }

    STATIC
    VOID
    SendBlock ()
    /*++
    Routine Description:

      Huffman code the block and output it.

    Argument: (VOID)

    Returns: (VOID)

    --*/
    {
      UINT32 i, k, Flags, Root, Pos, Size;
      Flags = 0;

      Root = MakeTree(NC, mCFreq, mCLen, mCCode);
      Size = mCFreq[Root];
      PutBits(16, Size);
      if (Root >= NC) {
        CountTFreq();
        Root = MakeTree(NT, mTFreq, mPTLen, mPTCode);
        if (Root >= NT) {
          WritePTLen(NT, TBIT, 3);
        } else {
          PutBits(TBIT, 0);
          PutBits(TBIT, Root);
        }
        WriteCLen();
      } else {
        PutBits(TBIT, 0);
        PutBits(TBIT, 0);
        PutBits(CBIT, 0);
        PutBits(CBIT, Root);
      }
      Root = MakeTree(NP, mPFreq, mPTLen, mPTCode);
      if (Root >= NP) {
        WritePTLen(NP, PBIT, -1);
      } else {
        PutBits(PBIT, 0);
        PutBits(PBIT, Root);
      }
      Pos = 0;
      for (i = 0; i < Size; i++) {
        if (i % UINT8_BIT == 0) {
          Flags = mBuf[Pos++];
        } else {
          Flags <<= 1;
        }
        if (Flags & (1U << (UINT8_BIT - 1))) {
          EncodeC(mBuf[Pos++] + (1U << UINT8_BIT));
          k = mBuf[Pos++] << UINT8_BIT;
          k += mBuf[Pos++];
          EncodeP(k);
        } else {
          EncodeC(mBuf[Pos++]);
        }
      }
      for (i = 0; i < NC; i++) {
        mCFreq[i] = 0;
      }
      for (i = 0; i < NP; i++) {
        mPFreq[i] = 0;
      }
    }


    STATIC
    VOID
    Output (
      IN UINT32 c,
      IN UINT32 p
      )
    /*++

    Routine Description:

      Outputs an Original Character or a Pointer

    Arguments:

      c     - The original character or the 'String Length' element of a Pointer
      p     - The 'Position' field of a Pointer

    Returns: (VOID)

    --*/
    {
      STATIC UINT32 CPos;

      if ((mOutputMask >>= 1) == 0) {
        mOutputMask = 1U << (UINT8_BIT - 1);
        if (mOutputPos >= mBufSiz - 3 * UINT8_BIT) {
          SendBlock();
          mOutputPos = 0;
        }
        CPos = mOutputPos++;
        mBuf[CPos] = 0;
      }
      mBuf[mOutputPos++] = (UINT8) c;
      mCFreq[c]++;
      if (c >= (1U << UINT8_BIT)) {
        mBuf[CPos] |= mOutputMask;
        mBuf[mOutputPos++] = (UINT8)(p >> UINT8_BIT);
        mBuf[mOutputPos++] = (UINT8) p;
        c = 0;
        while (p) {
          p >>= 1;
          c++;
        }
        mPFreq[c]++;
      }
    }
```

```
STATIC
VOID
HufEncodeStart ()
{
  INT32 i;

  for (i = 0; i < NC; i++) {
    mCFreq[i] = 0;
  }
  for (i = 0; i < NP; i++) {
    mPFreq[i] = 0;
  }
  mOutputPos = mOutputMask = 0;
  InitPutBits();
  return;
}

STATIC
VOID
HufEncodeEnd ()
{
  SendBlock();

  //
  // Flush remaining bits
  //
  PutBits(UINT8_BIT - 1, 0);

  return;
}


STATIC
VOID
MakeCrcTable ()
{
  UINT32 i, j, r;

  for (i = 0; i <= UINT8_MAX; i++) {
    r = i;
    for (j = 0; j < UINT8_BIT; j++) {
      if (r & 1) {
        r = (r >> 1) ^ CRCPOLY;
      } else {
        r >>= 1;
      }
    }
    mCrcTable[i] = (UINT16)r;
  }
}

STATIC
VOID
PutBits (
  IN INT32 n,
  IN UINT32 x
  )
/*++
Routine Description:

  Outputs rightmost n bits of x

Argments:

  n   - the rightmost n bits of the data is used
  x   - the data

Returns: (VOID)

--*/
{
  UINT8 Temp;

  if (n < mBitCount) {
    mSubBitBuf |= x << (mBitCount -= n);
  } else {

    Temp = (UINT8)(mSubBitBuf | (x >> (n -= mBitCount)));
    if (mDst < mDstUpperLimit) {
      *mDst++ = Temp;
    }
    mCompSize++;

    if (n < UINT8_BIT) {
      mSubBitBuf = x << (mBitCount = UINT8_BIT - n);
    } else {

      Temp = (UINT8)(x >> (n - UINT8_BIT));
      if (mDst < mDstUpperLimit) {
        *mDst++ = Temp;
      }
      mCompSize++;

      mSubBitBuf = x << (mBitCount = 2 * UINT8_BIT - n);
    }
  }
}

STATIC
INT32
FreadCrc (
  OUT UINT8 *p,
  IN  INT32 n
  )
/*++

Routine Description:

  Read in source data

Arguments:

  p   - the buffer to hold the data
  n   - number of bytes to read
```

```
Returns:

  number of bytes actually read

--*/
{
  INT32 i;

  for (i = 0; mSrc < mSrcUpperLimit && i < n; i++) {
    *p++ = *mSrc++;
  }
  n = i;

  p -= n;
  mOrigSize += n;
  while (--i >= 0) {
    UPDATE_CRC(*p++);
  }
  return n;
}


STATIC
VOID
InitPutBits ()
{
  mBitCount = UINT8_BIT;
  mSubBitBuf = 0;
}

STATIC
VOID
CountLen (
  IN INT32 i
  )
/*++

Routine Description:

  Count the number of each code length for a Huffman tree.

Arguments:

  i   - the top node

Returns: (VOID)

--*/
{
  STATIC INT32 Depth = 0;

  if (i < mN) {
    mLenCnt[(Depth < 16) ? Depth : 16]++;
  } else {
    Depth++;
    CountLen(mLeft [i]);
    CountLen(mRight[i]);
    Depth--;
  }
}
```

```
STATIC
VOID
MakeLen (
  IN INT32 Root
  )
/*++

Routine Description:

  Create code length array for a Huffman tree

Arguments:

  Root   - the root of the tree

--*/
{
  INT32 i, k;
  UINT32 Cum;

  for (i = 0; i <= 16; i++) {
    mLenCnt[i] = 0;
  }
  CountLen(Root);

  //
  // Adjust the length count array so that
  // no code will be generated longer than the designated length
  //

  Cum = 0;
  for (i = 16; i > 0; i--) {
    Cum += mLenCnt[i] << (16 - i);
  }
  while (Cum != (1U << 16)) {
    mLenCnt[16]--;
    for (i = 15; i > 0; i--) {
      if (mLenCnt[i] != 0) {
        mLenCnt[i]--;
        mLenCnt[i+1] += 2;
        break;
      }
    }
    Cum--;
  }
  for (i = 16; i > 0; i--) {
    k = mLenCnt[i];
    while (--k >= 0) {
      mLen[*mSortPtr++] = (UINT8)i;
    }
  }
}
```

```
STATIC
VOID
DownHeap (
  IN INT32 i
  )
{
  INT32 j, k;

  //
  // priority queue: send i-th entry down heap
  //

  k = mHeap[i];
  while ((j = 2 * i) <= mHeapSize) {
    if (j < mHeapSize && mFreq[mHeap[j]] > mFreq[mHeap[j + 1]]) {
      j++;
    }
    if (mFreq[k] <= mFreq[mHeap[j]]) {
      break;
    }
    mHeap[i] = mHeap[j];
    i = j;
  }
  mHeap[i] = (INT16)k;
}

STATIC
VOID
MakeCode (
  IN  INT32 n,
  IN  UINT8 Len[],
  OUT UINT16 Code[]
  )
/*++

Routine Description:

  Assign code to each symbol based on the code length array

Arguments:

  n     - number of symbols
  Len   - the code length array
  Code  - stores codes for each symbol

Returns: (VOID)

--*/
{
  INT32    i;
  UINT16   Start[18];

  Start[1] = 0;
  for (i = 1; i <= 16; i++) {
    Start[i + 1] = (UINT16)((Start[i] + mLenCnt[i]) << 1);
  }
  for (i = 0; i < n; i++) {
    Code[i] = Start[Len[i]]++;
  }
}
```

```
STATIC
INT32
MakeTree (
  IN  INT32   NParm,
  IN  UINT16  FreqParm[],
  OUT UINT8   LenParm[],
  OUT UINT16  CodeParm[]
  )
/*++

Routine Description:

  Generates Huffman codes given a frequency distribution of symbols

Arguments:

  NParm    - number of symbols
  FreqParm - frequency of each symbol
  LenParm  - code length for each symbol
  CodeParm - code for each symbol

Returns:

  Root of the Huffman tree.

--*/
{
  INT32 i, j, k, Avail;

  //
  // make tree, calculate len[], return root
  //

  mN = NParm;
  mFreq = FreqParm;
  mLen = LenParm;
  Avail = mN;
  mHeapSize = 0;
  mHeap[1] = 0;
  for (i = 0; i < mN; i++) {
    mLen[i] = 0;
    if (mFreq[i]) {
      mHeap[++mHeapSize] = (INT16)i;
    }
  }
  if (mHeapSize < 2) {
    CodeParm[mHeap[1]] = 0;
    return mHeap[1];
  }
  for (i = mHeapSize / 2; i >= 1; i--) {

    //
    // make priority queue
    //
    DownHeap(i);
  }
  mSortPtr = CodeParm;
  do {
    i = mHeap[1];
```

```
      if (i < mN) {
        *mSortPtr++ = (UINT16)i;
      }
      mHeap[1] = mHeap[mHeapSize--];
      DownHeap(1);
      j = mHeap[1];
      if (j < mN) {
        *mSortPtr++ = (UINT16)j;
      }
      k = Avail++;
      mFreq[k] = (UINT16)(mFreq[i] + mFreq[j]);
      mHeap[1] = (INT16)k;
      DownHeap(1);
      mLeft[k] = (UINT16)i;
      mRight[k] = (UINT16)j;
    } while (mHeapSize > 1);

    mSortPtr = CodeParm;
    MakeLen(k);
    MakeCode(NParm, LenParm, CodeParm);

    //
    // return root
    //
    return k;
  }
```

Appendix I
Decompression Source Code

```
/*++

Copyright (c) 2001-2002  Intel Corporation

Module Name:

  Decompress.c

Abstract:

  Decompressor.

--*/

#include "EfiCommon.h"


#define     BITBUFSIZ         16
#define     WNDBIT            13
#define     WNDSIZ            (1U << WNDBIT)
#define     MAXMATCH          256
#define     THRESHOLD         3
#define     CODE_BIT          16
#define     UINT8_MAX         0xff
#define     BAD_TABLE         -1

//
// C: Char&Len Set; P: Position Set; T: exTra Set
//

#define     NC                (0xff + MAXMATCH + 2 - THRESHOLD)
#define     CBIT              9
#define     NP                (WNDBIT + 1)
#define     NT                (CODE_BIT + 3)
#define     PBIT              4
#define     TBIT              5
#if NT > NP
  #define     NPT               NT
#else
  #define     NPT               NP
#endif


typedef struct {
  UINT8       *mSrcBase;      //Starting address of compressed data
  UINT8       *mDstBase;      //Starting address of decompressed data

  UINT16      mBytesRemain;
  UINT16      mBitCount;
  UINT16      mBitBuf;
  UINT16      mSubBitBuf;
  UINT16      mBufSiz;
  UINT16      mBlockSize;
  UINT32      mDataIdx;
  UINT32      mCompSize;
  UINT32      mOrigSize;
  UINT32      mOutBuf;
  UINT32      mInBuf;

  UINT16      mBadTableFlag;

  UINT8       mBuffer[WNDSIZ];
  UINT16      mLeft[2 * NC - 1];
  UINT16      mRight[2 * NC - 1];
  UINT32      mBuf;
  UINT8       mCLen[NC];
  UINT8       mPTLen[NPT];
  UINT16      mCTable[4096];
  UINT16      mPTTable[256];
} SCRATCH_DATA;


//
// Function Prototypes
//

STATIC
VOID
FillBuf (
  IN  SCRATCH_DATA  *Sd,
  IN  UINT16        NumOfBits
  );

STATIC
VOID
Decode (
  SCRATCH_DATA  *Sd,
  UINT16        NumOfBytes
  );


//
// Functions
//

EFI_STATUS
EFIAPI
GetInfo (
  IN      EFI_DECOMPRESS_PROTOCOL  *This,
  IN      VOID                     *Source,
  IN      UINT32                   SrcSize,
  OUT     UINT32                   *DstSize,
  OUT     UINT32                   *ScratchSize
  )
/*++

Routine Description:

  The implementation of EFI_DECOMPRESS_PROTOCOL.GetInfo().
```

```
Arguments:

  This        - Protocol instance pointer.
  Source      - The source buffer containing the compressed data.
  SrcSize     - The size of source buffer
  DstSize     - The size of destination buffer.
  ScratchSize - The size of scratch buffer.

Returns:

  EFI_SUCCESS           - The size of destination buffer and the size of
scratch buffer are successull retrieved.
  EFI_INVALID_PARAMETER - The source data is corrupted

--*/
{
  UINT8 *Src;

  *ScratchSize = sizeof (SCRATCH_DATA);

  Src = Source;
  if (SrcSize < 8) {
    return EFI_INVALID_PARAMETER;
  }

  *DstSize = Src[4] + (Src[5] << 8) + (Src[6] << 16) + (Src[7] << 24);
  return EFI_SUCCESS;
}


EFI_STATUS
EFIAPI
Decompress (
  IN      EFI_DECOMPRESS_PROTOCOL *This,
  IN      VOID                    *Source,
  IN      UINT32                  SrcSize,
  IN OUT  VOID                    *Destination,
  IN      UINT32                  DstSize,
  IN OUT  VOID                    *Scratch,
  IN      UINT32                  ScratchSize
  )
/*++

Routine Description:

  The implementation of EFI_DECOMPRESS_PROTOCOL.Decompress().

Arguments:

  This        - The protocol instance.
  Source      - The source buffer containing the compressed data.
  SrcSize     - The size of the source buffer
  Destination - The destination buffer to store the decompressed data
  DstSize     - The size of the destination buffer.
  Scratch     - The buffer used internally by the decompress routine.  This
buffer is needed to store intermediate data.
  ScratchSize - The size of scratch buffer.
```

```
Returns:

  EFI_SUCCESS           - Decompression is successfull
  EFI_INVALID_PARAMETER - The source data is corrupted

--*/
{
  UINT32        Index;
  UINT16        Count;
  UINT32        CompSize;
  UINT32        OrigSize;
  UINT8         *Dst1;
  EFI_STATUS    Status;
  SCRATCH_DATA  *Sd;
  UINT8         *Src;
  UINT8         *Dst;

  Status = EFI_SUCCESS;
  Src   = Source;
  Dst   = Destination;
  Dst1 = Dst;

  if (ScratchSize < sizeof (SCRATCH_DATA)) {
      return  EFI_INVALID_PARAMETER;
  }

  Sd = (SCRATCH_DATA *)Scratch;

  if (SrcSize < 8) {
    return EFI_INVALID_PARAMETER;
  }

  CompSize = Src[0] + (Src[1] << 8) + (Src[2] << 16) + (Src[3] << 24);
  OrigSize = Src[4] + (Src[5] << 8) + (Src[6] << 16) + (Src[7] << 24);

  if (SrcSize < CompSize + 8) {
    return EFI_INVALID_PARAMETER;
  }

  Src = Src + 8;

  for (Index = 0; Index < sizeof(SCRATCH_DATA); Index++) {
    ((UINT8*)Sd)[Index] = 0;
  }

  Sd->mBytesRemain = (UINT16)(-1);
  Sd->mSrcBase = Src;
  Sd->mDstBase = Dst;
  Sd->mCompSize = CompSize;
  Sd->mOrigSize = OrigSize;

  //
  // Fill the first two bytes
  //
  FillBuf(Sd, BITBUFSIZ);

  while (Sd->mOrigSize > 0) {

    Count = (UINT16) (WNDSIZ < Sd->mOrigSize? WNDSIZ: Sd->mOrigSize);
    Decode (Sd, Count);
    if (Sd->mBadTableFlag != 0) {
      //
      // Something wrong with the source
      //
      return EFI_INVALID_PARAMETER;
    }

    for (Index = 0; Index < Count; Index ++) {
      if (Dst1 < Dst + DstSize) {
        *Dst1++ = Sd->mBuffer[Index];
      } else {
        return EFI_INVALID_PARAMETER;
      }
    }

    Sd->mOrigSize -= Count;
  }

  if (Sd->mBadTableFlag != 0) {
    Status = EFI_INVALID_PARAMETER;
  } else {
    Status = EFI_SUCCESS;
  }

  return  Status;
}


STATIC
VOID
FillBuf (
  IN  SCRATCH_DATA  *Sd,
  IN  UINT16        NumOfBits
  )
/*++

Routine Description:

  Shift mBitBuf NumOfBits left.  Read in NumOfBits of bits from source.

Arguments:

  Sd         - The global scratch data
  NumOfBit   - The number of bits to shift and read.

Returns: (VOID)

--*/
{
  Sd->mBitBuf = (UINT16)(Sd->mBitBuf << NumOfBits);

  while (NumOfBits > Sd->mBitCount) {

    Sd->mBitBuf |= (UINT16)(Sd->mSubBitBuf <<
      (NumOfBits = (UINT16)(NumOfBits - Sd->mBitCount)));

    if (Sd->mCompSize > 0) {
      //
      // Get 1 byte into SubBitBuf
      //
      Sd->mCompSize --;
      Sd->mSubBitBuf = 0;
      Sd->mSubBitBuf = Sd->mSrcBase[Sd->mInBuf ++];
      Sd->mBitCount = 8;

    } else {

      Sd->mSubBitBuf = 0;
      Sd->mBitCount = 8;

    }
  }

  Sd->mBitCount = (UINT16)(Sd->mBitCount - NumOfBits);
  Sd->mBitBuf |= Sd->mSubBitBuf >> Sd->mBitCount;
}


STATIC
UINT16
GetBits(
  IN  SCRATCH_DATA  *Sd,
  IN  UINT16    NumOfBits
  )
/*++

Routine Description:

  Get NumOfBits of bits out from mBitBuf.  Fill mBitBuf with subsequent
  NumOfBits of bits from source.  Returns NumOfBits of bits that are
  popped out.

Arguments:

  Sd             - The global scratch data.
  NumOfBits      - The number of bits to pop and read.

Returns:

  The bits that are popped out.

--*/
{
  UINT16  OutBits;

  OutBits = (UINT16)(Sd->mBitBuf >> (BITBUFSIZ - NumOfBits));

  FillBuf (Sd, NumOfBits);

  return  OutBits;
}
```

```
STATIC
UINT16
MakeTable (
  IN  SCRATCH_DATA  *Sd,
  IN  UINT16      NumOfChar,
  IN  UINT8       *BitLen,
  IN  UINT16      TableBits,
  OUT UINT16       *Table
  )
/*++

Routine Description:

  Creates Huffman Code mapping table according to code length array.

Arguments:

  Sd         - The global scratch data
  NumOfChar - Number of symbols in the symbol set
  BitLen     - Code length array
  TableBits - The width of the mapping table
  Table      - The table

Returns:

  0          - OK.
  BAD_TABLE - The table is corrupted.

--*/
{
  UINT16  Count[17];
  UINT16  Weight[17];
  UINT16  Start[18];
  UINT16   *p;
  UINT16  k;
  UINT16  i;
  UINT16  Len;
  UINT16  Char;
  UINT16  JuBits;
  UINT16  Avail;
  UINT16  NextCode;
  UINT16  Mask;


  for (i = 1; i <= 16; i ++) {
    Count[i] = 0;
  }

  for (i = 0; i < NumOfChar; i++) {
    Count[BitLen[i]]++;
  }

  Start[1] = 0;

  for (i = 1; i <= 16; i ++) {
    Start[i + 1] = (UINT16)(Start[i] + (Count[i] << (16 - i)));
  }
```

```
    if (Start[17] != 0) {/*(1U << 16)*/
      return (UINT16)BAD_TABLE;
    }

    JuBits = (UINT16)(16 - TableBits);

    for (i = 1; i <= TableBits; i ++) {
      Start[i] >>= JuBits;
      Weight[i] = (UINT16)(1U << (TableBits - i));
    }

    while (i <= 16) {
      Weight[i++] = (UINT16)(1U << (16 - i));
    }

    i = (UINT16)(Start[TableBits + 1] >> JuBits);

    if (i != 0) {
      k = (UINT16)(1U << TableBits);
      while (i != k) {
        Table[i++] = 0;
      }
    }

    Avail = NumOfChar;
    Mask = (UINT16)(1U << (15 - TableBits));

    for (Char = 0; Char < NumOfChar; Char++) {

      Len = BitLen[Char];
      if (Len == 0) {
        continue;
      }

      NextCode = (UINT16)(Start[Len] + Weight[Len]);

      if (Len <= TableBits) {

        for (i = Start[Len]; i < NextCode; i ++) {
          Table[i] = Char;
        }

      } else {

        k = Start[Len];
        p = &Table[k >> JuBits];
        i = (UINT16)(Len - TableBits);

        while (i != 0) {
          if (*p == 0) {
            Sd->mRight[Avail] = Sd->mLeft[Avail] = 0;
            *p = Avail ++;
          }

          if (k & Mask) {
            p = &Sd->mRight[*p];
          } else {
            p = &Sd->mLeft[*p];
          }
          k <<= 1;
          i --;
        }

        *p = Char;

      }

      Start[Len] = NextCode;
    }

    //
    // Succeeds
    //
    return 0;
  }


  STATIC
  UINT16
  DecodeP (
    IN  SCRATCH_DATA  *Sd
    )
  /*++

  Routine description:

    Decodes a position value.

  Arguments:

    Sd      - the global scratch data

  Returns:

    The position value decoded.

  --*/
  {
    UINT16  Val;
    UINT16  Mask;

    Val = Sd->mPTTable[Sd->mBitBuf >> (BITBUFSIZ - 8)];

    if (Val >= NP) {
      Mask = 1U << (BITBUFSIZ - 1 - 8);

      do {

        if (Sd->mBitBuf & Mask) {
          Val = Sd->mRight[Val];
        } else {
          Val = Sd->mLeft[Val];
        }

        Mask >>= 1;
      } while (Val >= NP);
    }
    //
    // Advance what we have read
    //
    FillBuf (Sd, Sd->mPTLen[Val]);

    if (Val) {
      Val = (UINT16)((1U << (Val - 1)) + GetBits (Sd, (UINT16)(Val - 1)));
    }

    return Val;
  }


  STATIC
  UINT16
  ReadPTLen (
    IN  SCRATCH_DATA  *Sd,
    IN  UINT16  nn,
    IN  UINT16  nbit,
    IN  UINT16  Special
    )
  /*++

  Routine Descriptiion:

    Reads code lengths for the Extra Set or the Position Set

  Arguments:

    Sd        - The global scratch data
    nn        - Number of symbols
    nbit      - Number of bits needed to represent nn
    Special   - The special symbol that needs to be taken care of

  Returns:

    0         - OK.
    BAD_TABLE - Table is corrupted.

  --*/
  {
    UINT16    n;
    UINT16    c;
    UINT16    i;
    UINT16    Mask;

    n = GetBits (Sd, nbit);

    if (n == 0) {
      c = GetBits (Sd, nbit);

      for ( i = 0; i < 256; i ++) {
        Sd->mPTTable[i] = c;
      }

      for ( i = 0; i < nn; i++) {
        Sd->mPTLen[i] = 0;
      }
      return 0;
    }

    i = 0;

    while (i < n) {

      c = (UINT16)(Sd->mBitBuf >> (BITBUFSIZ - 3));

      if (c == 7) {
        Mask = 1U << (BITBUFSIZ - 1 - 3);
        while (Mask & Sd->mBitBuf) {
          Mask >>= 1;
          c += 1;
        }
      }

      FillBuf (Sd, (UINT16)((c < 7) ? 3 : c - 3));

      Sd->mPTLen [i++] = (UINT8)c;

      if (i == Special) {
        c = GetBits (Sd, 2);
        while ((INT16)(--c) >= 0) {
          Sd->mPTLen[i++] = 0;
        }
      }
    }

    while (i < nn) {
      Sd->mPTLen [i++] = 0;
    }

    return ( MakeTable (Sd, nn, Sd->mPTLen, 8, Sd->mPTTable) );
  }


  STATIC
  VOID
  ReadCLen (
    SCRATCH_DATA  *Sd
    )
  /*++

  Routine Description:

    Reads code lengths for Char&Len Set.

  Arguments:

    Sd    - the global scratch data

  Returns: (VOID)

  --*/
  {
    UINT16    n;
    UINT16    c;
    UINT16    i;
    UINT16    Mask;
```

```
n = GetBits(Sd, CBIT);

if (n == 0) {
  c = GetBits(Sd, CBIT);

  for (i = 0; i < NC; i ++) {
    Sd->mCLen[i] = 0;
  }

  for (i = 0; i < 4096; i ++) {
    Sd->mCTable[i] = c;
  }

  return;
}

i = 0;
while (i < n) {

  c = Sd->mPTTable[Sd->mBitBuf >> (BITBUFSIZ - 8)];
  if (c >= NT) {
    Mask = 1U << (BITBUFSIZ - 1 - 8);

    do {

      if (Mask & Sd->mBitBuf) {
        c = Sd->mRight [c];
      } else {
        c = Sd->mLeft [c];
      }

      Mask >>= 1;

    }while (c >= NT);
  }

  //
  // Advance what we have read
  //
  FillBuf (Sd, Sd->mPTLen[c]);

  if (c <= 2) {

    if (c == 0) {
      c = 1;
    } else if (c == 1) {
      c = (UINT16)(GetBits (Sd, 4) + 3);
    } else if (c == 2) {
      c = (UINT16)(GetBits (Sd, CBIT) + 20);
    }

    while ((INT16)(--c) >= 0) {
      Sd->mCLen[i++] = 0;
    }

  } else {
        Sd->mCLen[i++] = (UINT8)(c - 2);

      }
    }

    while (i < NC) {
      Sd->mCLen[i++] = 0;
    }

    MakeTable (Sd, NC, Sd->mCLen, 12, Sd->mCTable);

    return;
  }


  STATIC
  UINT16
  DecodeC (
    SCRATCH_DATA  *Sd
    )
  /*++

  Routine Description:

    Decode a character/length value.

  Arguments:

    Sd    - The global scratch data.

  Returns:

    The value decoded.

  --*/
  {
    UINT16      j;
    UINT16      Mask;

    if (Sd->mBlockSize == 0) {

      //
      // Starting a new block
      //

      Sd->mBlockSize = GetBits(Sd, 16);
      Sd->mBadTableFlag = ReadPTLen (Sd, NT, TBIT, 3);
      if (Sd->mBadTableFlag != 0) {
        return 0;
      }

      ReadCLen (Sd);

      Sd->mBadTableFlag = ReadPTLen (Sd, NP, PBIT, (UINT16)(-1));
      if (Sd->mBadTableFlag != 0) {
        return 0;
      }
    }
    Sd->mBlockSize --;
    j = Sd->mCTable[Sd->mBitBuf >> (BITBUFSIZ - 12)];

    if (j >= NC) {
      Mask = 1U << (BITBUFSIZ - 1 - 12);

      do {
        if (Sd->mBitBuf & Mask) {
          j = Sd->mRight[j];
        } else {
          j = Sd->mLeft[j];
        }

        Mask >>= 1;
      } while (j >= NC);
    }

    //
    // Advance what we have read
    //
    FillBuf(Sd, Sd->mCLen[j]);

    return j;
  }


  STATIC
  VOID
  Decode (
    SCRATCH_DATA  *Sd,
    UINT16        NumOfBytes
    )
   /*++

  Routine Description:

    Decode NumOfBytes and put the resulting data at starting point of mBuffer.
    The buffer is circular.

  Arguments:

    Sd              - The global scratch data
    NumOfBytes      - Number of bytes to decode

  Returns: (VOID)

   --*/
  {
    UINT16      di;
    UINT16      r;
    UINT16      c;

    r = 0;
    di = 0;

    Sd->mBytesRemain --;
    while ((INT16)(Sd->mBytesRemain) >= 0) {
      Sd->mBuffer[di++] = Sd->mBuffer[Sd->mDataIdx++];
```

```
      if (Sd->mDataIdx >= WNDSIZ) {
        Sd->mDataIdx -= WNDSIZ;
      }

      r ++;
      if (r >= NumOfBytes) {
        return;
      }
      Sd->mBytesRemain --;
    }

    for (;;) {
      c = DecodeC (Sd);
      if (Sd->mBadTableFlag != 0) {
        return;
      }

      if (c < 256) {

        //
        // Process an Original character
        //

        Sd->mBuffer[di++] = (UINT8)c;
        r ++;
        if (di >= WNDSIZ) {
          return;
        }

      } else {

        //
        // Process a Pointer
        //

        c = (UINT16)(c - (UINT8_MAX + 1 - THRESHOLD));
        Sd->mBytesRemain = c;

        Sd->mDataIdx = (r - DecodeP(Sd) - 1) & (WNDSIZ - 1); //Make circular

        di = r;

        Sd->mBytesRemain --;
        while ((INT16)(Sd->mBytesRemain) >= 0) {
          Sd->mBuffer[di++] = Sd->mBuffer[Sd->mDataIdx++];
          if (Sd->mDataIdx >= WNDSIZ) {
            Sd->mDataIdx -= WNDSIZ;
          }

          r ++;
          if (di >= WNDSIZ) {
            return;
          }
          Sd->mBytesRemain --;
        }
      }
    }

    return;
  }
```

Appendix J EFI Byte Code Virtual Machine Opcode Summary

The following table lists the opcodes for EBC instructions. Note that opcodes only require 6 bits of the opcode byte of EBC instructions. The other two bits are used for other encodings that are dependent on the particular instruction.

Table J-1. EBC Virtual Machine Opcode Summary

Opcode	Description
0x00	BREAK [break code]
0x01	JMP32{cs|cc} {@}R_1 {Immed32|Index32} JMP64{cs|cc} Immed64
0x02	JMP8{cs|cc} Immed8
0x03	CALL32{EX}{a} {@}R_1 {Immed32|Index32} CALL64{EX}{a} Immed64
0x04	RET
0x05	CMP[32|64]eq R_1, {@}R_2 {Index16|Immed16}
0x06	CMP[32|64]lte R_1, {@}R_2 {Index16|Immed16}
0x07	CMP[32|64]gte R_1, {@}R_2 {Index16|Immed16}
0x08	CMP[32|64]ulte R_1, {@}R_2 {Index16|Immed16}
0x09	CMP[32|64]ugte R_1, {@}R_2 {Index16|Immed16}
0x0A	NOT[32|64] {@}R_1, {@}R_2 {Index16|Immed16}
0x0B	NEG[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x0C	ADD[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x0D	SUB[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x0E	MUL[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x0F	MULU[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x10	DIV[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x11	DIVU[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x12	MOD[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x13	MODU[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x14	AND[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x15	OR[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x16	XOR[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x17	SHL[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x18	SHR[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x19	ASHR[32|64] {@}R_1,{@}R_2 {Index16|Immed16}

continued

Table J-1. EBC Virtual Machine Opcode Summary (continued)

Opcode	Description
0x1A	EXTNDB[32|64] {@}R_1, {@}R_2{Index16|Immed16}
0x1B	EXTNDW[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x1C	EXTNDD[32|64] {@}R_1,{@}R_2 {Index16|Immed16}
0x1D	MOVbw {@}R_1 {Index16}, {@}R_2 {Index16}
0x1E	MOVww {@}R_1 {Index16}, {@}R_2 {Index16}
0x1F	MOVdw {@}R_1 {Index16}, {@}R_2 {Index16}
0x20	MOVqw {@}R_1 {Index16}, {@}R_2 {Index16}
0x21	MOVbd {@}R_1 {Index32}, {@}R_2 {Index32}
0x22	MOVwd {@}R_1 {Index32}, {@}R_2 {Index32}
0x23	MOVdd {@}R_1 {Index32}, {@}R_2 {Index32}
0x24	MOVqd {@}R_1 {Index32}, {@}R_2 {Index32}
0x25	MOVsnw {@}R_1 {Index16}, {@}R_2 {Index16|Immed16}
0x26	MOVsnd {@}R_1 {Index32}, {@}R_2{Index32|Immed32}
0x27	Reserved
0x28	MOVqq {@}R_1 {Index64}, {@}R_2 {Index64}
0x29	LOADSP [Flags], R_2
0x2A	STORESP R_1, [IP|Flags]
0x2B	PUSH[32|64] {@}R_1{Index16|Immed16}
0x2C	POP[32|64] {@}R_1{Index16|Immed16}
0x2D	CMPI[32|64][w|d]eq {@}R_1 {Index16}, Immed16|Immed32
0x2E	CMPI[32|64][w|d]lte {@}R_1 {Index16}, Immed16|Immed32
0x2F	CMPI[32|64][w|d]gte {@}R_1 {Index16}, Immed16|Immed32
0x30	CMPI[32|64][w|d]ulte {@}R_1 {Index16}, Immed16|Immed32
0x31	CMPI[32|64][w|d]ugte {@}R_1 {Index16}, Immed16|Immed32
0x32	MOVnw {@}R_1 {Index16}, {@}R_2{Index16}
0x33	MOVnd {@}R_1 {Index32}, {@}R_2{Index32}
0x34	Reserved
0x35	PUSHn {@}R_1{Index16|Immed16}
0x36	POPn {@}R_1{Index16|Immed16}
0x37	MOVI[b|w|d|q][w|d|q] {@}R_1{Index16}, Immed16|32|64
0x38	MOVIn[w|d|q] {@}R_1{Index16}, Index16|32|64
0x39	MOVREL[w|d|q] {@}R_1{Index16}, Immed16|32|64
0x3A	Reserved
0x3B	Reserved
0x3C	Reserved
0x3D	Reserved
0x3E	Reserved
0x3F	Reserved

Appendix K
Alphabetic Function Lists

This appendix contains two tables that list all EFI functions alphabetically. Table K-1 lists the functions in pure alphabetic order. Functions that have the same name can be distinguished by the associated service or protocol (column 2). For example, there are two "Flush" functions, one from the Device I/O Protocol and one from the File System Protocol. Table K-2 orders the functions alphabetically within a service or protocol. That is, column one names the service or protocol, and column two lists the functions in the service or protocol.

Table K-1. Functions Listed in Alphabetic Order

Function Name	Service or Protocol	Subservice	Function Description
AllocateBuffer	Device I/O Protocol		Allocates pages that are suitable for a common buffer mapping.
AllocateBuffer	PCI I/O Protocol		Allocates pages that are suitable for a common buffer mapping.
AllocateBuffer	PCI Root Bridge I/O Protocol		Allocates pages that are suitable for a common buffer mapping.
AllocatePages	Boot Services	Memory Allocation Services	Allocates memory pages of a particular type.
AllocatePool	Boot Services	Memory Allocation Services	Allocates pool of a particular type.
Arp	PXE Base Code Protocol		Uses the ARP protocol to resolve a MAC address.
AsyncInterruptTransfer	USB Host Controller Protocol		Submits an asynchronous interrupt transfer to an interrupt endpoint of a USB device.
AsyncIsochronousTransfer	USB Host Controller Protocol		Submits nonblocking USB isochronous transfer.
Attributes	PCI I/O Protocol		Performs an operation on the attributes that this PCI controller supports.
Blt	UGA Draw Protocol		Blt a rectangle of pixels on the graphics screen. Blt stands for BLock Transfer.
BuildDevicePath	SCSI Passthru Protocol		Used to allocate and build a device path node for a SCSI device on a SCSI channel.
BulkTransfer	USB Host Controller Protocol		Submits a bulk transfer to a bulk endpoint of a USB device.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
CalculateCrc32	Boot Services	Miscellaneous Services	Computes and returns a 32-bit CRC for a data buffer.
Callback	PXE Base Code Callback Protocol		Callback routine used by the PXE Base Code `Dhcp()`, `Discover()`, `Mtftp()`, `UdpWrite()`, and `Arp()` functions.
CheckEvent	Boot Services	Event Services	Checks whether an event is in the signaled state.
ClearRootHubPortFeature	USB Host Controller Protocol		Clears the feature for the specified root hub port.
ClearScreen	Simple Text Output Protocol		Clears the screen with the currently set background color.
Close	File System Protocol		Closes the current file handle.
CloseEvent	Boot Services	Event Services	Closes and frees an event structure.
CloseProtocol	Boot Services	Protocol Handler Services	Removes elements from the list of agents consuming a protocol interface.
Configuration	PCI Root Bridge I/O Protocol		Gets the current resource settings for this PCI root bridge
ConnectController	Boot Services	Protocol Handler Services	Uses a set of precedence rules to find the best set of drivers to manage a controller.
ControlTransfer	USB Host Controller Protocol		Submits a control transfer to a target USB device.
ConvertPointer	Runtime Services	Virtual Memory Services	Converts internal pointers when switching to virtual addressing.
CopyMem	Boot Services	Miscellaneous Services	Copies the contents of one buffer to another buffer.
CopyMem	PCI I/O Protocol		Allows one region of PCI memory space to be copied to another region of PCI memory space
CopyMem	PCI Root Bridge I/O Protocol		Allows one region of PCI root bridge memory space to be copied to another region of PCI root bridge memory space.
CreateDevice	UGA I/O Protocol		Dynamically allocate storage for a child `UGA_DEVICE`.
CreateEvent	Boot Services	Event Services	Creates a general-purpose event structure.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
CreateThunk	EFI Byte Code Protocol		Creates a thunk for an EBC image entry point or protocol service, and returns a pointer to the thunk.
Decompress	Decompress Protocol		Decompresses a compressed source buffer into an uncompressed destination buffer.
Delete	File System Protocol		Deletes a file.
DeleteDevice	UGA I/O Protocol		Free a dynamically allocated child **UGA_DEVICE** object that was allocated via **CreateDevice()**.
Dhcp	PXE Base Code Protocol		Attempts to complete a DHCPv4 D.O.R.A. (discover / offer / request / acknowledge) or DHCPv6 S.A.R.R (solicit / advertise / request / reply) sequence.
DisconnectController	Boot Services	Protocol Handler Services	Informs a set of drivers to stop managing a controller.
Discover	PXE Base Code Protocol		Attempts to complete the PXE Boot Server and/or boot image discovery sequence.
DispatchService	UGA I/O Protocol		This is the main UGA service dispatch routine for all **UGA_IO_REQUEST**s.
DriverLoaded	EFI Driver Override Protocol		Used to associate a driver image handle with a device path returned on a prior call.
EFI_IMAGE_ENTRY_POINT	Boot Services	Image Services	Prototype of an EFI Image's entry point.
EFI_PXE_BASE_CODE_CALLBACK	PXE Base Code Protocol		Callback function that is invoked when the PXE Base Code Protocol is waiting for an event.
EnableCursor	Simple Text Output Protocol		Turns the visibility of the cursor on/off.
Exit	Boot Services	Image Services	Exits the image's entry point.
ExitBootServices	Boot Services	Image Services	Terminates boot services.
FatToStr	Unicode Collation Protocol		Converts an 8.3 FAT file name in an OEM character set to a Null-terminated Unicode string.
Fill Header	UNDI Commands		This command is used to fill the media header(s) in transmit packet(s).

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
Flush	Device I/O Protocol		Flushes any posted write data to the device.
Flush	File System Protocol		Flushes all modified data associated with the file to the device.
Flush	PCI I/O Protocol		Flushes all PCI posted write transactions to system memory.
Flush	PCI Root Bridge I/O Protocol		Flushes all PCI posted write transactions to system memory.
FlushBlocks	Block I/O Protocol		Flushes any cached blocks.
ForceDefaults	EFI Driver Configuration Protocol		Forces a driver to set the default configuration options for a controller.
Free	Boot Integrity Services Protocol		Frees memory structures allocated and returned by other functions in the `EFI_BIS` protocol.
FreeBuffer	Device I/O Protocol		Frees pages that were allocated with `AllocateBuffer()`.
FreeBuffer	PCI I/O Protocol		Frees pages that were allocated with `AllocateBuffer()`.
FreeBuffer	PCI Root Bridge I/O Protocol		Free pages that were allocated with `AllocateBuffer()`.
FreePages	Boot Services	Memory Allocation Services	Frees memory pages.
FreePool	Boot Services	Memory Allocation Services	Frees allocated pool.
Get Config Info	UNDI Commands		This command is used to retrieve configuration information about the NIC being controlled by the UNDI.
Get Init Info	UNDI Commands		This command is used to retrieve initialization information that is needed by drivers and applications to initialized UNDI.
Get State	UNDI Commands		This command is used to determine the operational state of the UNDI.
Get Status	UNDI Commands		This command returns the current interrupt status and/or the transmitted buffer addresses.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
GetAttributes	PCI Root Bridge I/O Protocol		Gets the attributes that a PCI root bridge supports setting with `SetAttributes()`, and the attributes that a PCI root bridge is currently using.
GetBarAttributes	PCI I/O Protocol		Gets the attributes that this PCI controller supports setting on a BAR using `SetBarAttributes()`, and retrieves the list of resource descriptors for a BAR.
GetBootObjectAuthorizationCertificate	Boot Integrity Services Protocol		Retrieves the current digital certificate (if any) used by the `EFI_BIS` protocol as the source of authorization for verifying boot objects and altering configuration parameters
GetBootObjectAuthorizationCheckFlag	Boot Integrity Services Protocol		Retrieves the current setting of the authorization check flag that indicates whether or not authorization checks are required for boot objects.
GetBootObjectAuthorizationUpdateToken	Boot Integrity Services Protocol		Retrieves an uninterpreted token whose value gets included and signed in a subsequent request to alter the configuration parameters, to protect against attempts to "replay" such a request.
GetControl	Serial I/O Protocol		Reads the status of the control bits on a serial device.
GetControllerName	EFI Component Name Protocol		Retrieves a Unicode string that is the user readable name of the controller that is being managed by an EFI Driver.
GetDriver	EFI Bus-Specific Driver Override Protocol		Uses a bus-specific algorithm to retrieve a driver image handle for a controller.
GetDriver	EFI Driver Override Protocol		Retrieves the image handle of the platform override driver for a controller in the system.
GetDriverName	EFI Component Name Protocol		Retrieves a Unicode string that is the user readable name of the EFI Driver.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
GetDriverPath	EFI Driver Override Protocol		Retrieves the device path of the platform override driver for a controller in the system.
GetInfo	Decompress Protocol		Given the compressed source buffer, this function retrieves the size of the uncompressed destination buffer and the size of the scratch buffer required to perform the decompression.
GetInfo	File System Protocol		Gets the requested file or volume information.
GetLocation	PCI I/O Protocol		Retrieves this PCI controller's current PCI bus number, device number, and function number.
GetMaximumProcessor Index	Debug Support Protocol		Returns the maximum processor index value that may be used with `RegisterPeriodicCallback()` and `RegisterExceptionCallback()`
GetMemoryMap	Boot Services	Memory Allocation Services	Returns the current boot services memory map and memory map key.
GetMode	UGA Draw Protocol		Return the current frame buffer geometry and display refresh rate.
GetNextDevice	SCSI Passthru Protocol		Used to retrieve the list of legal Target IDs for the SCSI devices on a SCSI channel.
GetNextHighMonotonicCount	Runtime Services	Miscellaneous Services	Returns the next high 32 bits of a platform's monotonic counter.
GetNextMonotonicCount	Boot Services	Miscellaneous Services	Returns a monotonically increasing count for the platform.
GetNextVariableName	Runtime Services	Variable Services	Enumerates the current variable names.
GetPosition	File System Protocol		Returns the current file position.
GetRootHubPortNumber	USB Host Controller Protocol		Retrieves the number of root hub ports that are produced by the USB host controller.
GetRootHubPortStatus	USB Host Controller Protocol		Retrieves the status of the specified root hub port.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
GetSignatureInfo	Boot Integrity Services Protocol		Retrieves information about the digital signature algorithms supported and the identity of the installed authorization certificate, if any.
GetState	Simple Pointer Protocol		Retrieves the current state of a pointer device.
GetState	USB Host Controller Protocol		Retrieves the current state of the USB host controller.
GetStatus	Simple Network Protocol		Reads the current interrupt status and recycled transmit buffer status from the network interface.
GetTargetLun	SCSI Passthru Protocol		Used to translate a device path node to a Target ID and LUN.
GetTime	Runtime Services	Time Services	Returns the current time and date, and the time-keeping capabilities of the platform.
GetVariable	Runtime Services	Variable Services	Returns the value of the specific variable.
GetWakeupTime	Runtime Services	Time Services	Returns the current wakeup alarm clock setting.
HandleProtocol	Boot Services	Protocol Handler Services	Queries the list of protocol handlers on a device handle for the requested Protocol Interface.
Initialize	Boot Integrity Services Protocol		Initializes an application instance of the **EFI_BIS** protocol, returning a handle for the application instance.
Initialize	Simple Network Protocol		Resets the network adapter and allocates the transmit and receive buffers required by the network interface; also optionally allows space for additional transmit and receive buffers to be allocated
Initialize	UNDI Commands		This command resets the network adapter and initializes UNDI using the parameters supplied in the CPB.
InstallConfigurationTable	Boot Services	Miscellaneous Services	Adds, updates, or removes a configuration table from the EFI System Table.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
InstallMultipleProtocol Interfaces	Boot Services	Protocol Handler Services	Installs one or more protocol interfaces onto a handle.
InstallProtocolInterface	Boot Services	Protocol Handler Services	Adds a protocol interface to an existing or new device handle.
Interrupt Enables	UNDI Commands		The Interrupt Enables command can be used to read and/or change the current external interrupt enable settings.
InvalidateInstructionCache	Debug Support Protocol		Invalidate the instruction cache of the processor.
Io.Read	Device I/O Protocol		Reads from I/O ports on a bus.
Io.Read	PCI I/O Protocol		Allows BAR relative reads to PCI I/O space.
Io.Read	PCI Root Bridge I/O Protocol		Allows reads from I/O space.
Io.Write	Device I/O Protocol		Writes to I/O ports on a bus.
Io.Write	PCI I/O Protocol		Allows BAR relative writes to PCI I/O space.
Io.Write	PCI Root Bridge I/O Protocol		Allows writes to I/O space.
IsochronousTransfer	USB Host Controller Protocol		Submits isochronous transfer to an isochronous endpoint of a USB device.
LoadFile	Load File Protocol		Causes the driver to load the requested file.
LoadImage	Boot Services	Image Services	Function to dynamically load another EFI Image.
LocateDevicePath	Boot Services	Protocol Handler Services	Locates the closest handle that supports the specified protocol on the specified device path.
LocateHandle	Boot Services	Protocol Handler Services	Locates the handle(s) that support the specified protocol.
LocateHandleBuffer	Boot Services	Protocol Handler Services	Retrieves the list of handles from the handle database that meet the search criteria. The return buffer is automatically allocated.
LocateProtocol	Boot Services	Protocol Handler Services	Finds the first handle in the handle database the supports the requested protocol.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
Map	Device I/O Protocol		Provides the device specific addresses needed to access host memory for DMA.
Map	PCI I/O Protocol		Provides the PCI controller specific address needed to access system memory for DMA.
Map	PCI Root Bridge I/O Protocol		Provides the PCI controller specific addresses needed to access system memory for DMA.
MCast IP to MAC	UNDI Commands		Translate a multicast IPv4 or IPv6 address to a multicast MAC address.
MCastIPtoMAC	Simple Network Protocol		Allows a multicast IP address to be mapped to a multicast HW MAC address.
Mem.Read	Device I/O Protocol		Reads from memory on a bus.
Mem.Read	PCI I/O Protocol		Allows BAR relative reads to PCI memory space.
Mem.Read	PCI Root Bridge I/O Protocol		Allows reads from memory mapped I/O space.
Mem.Write	Device I/O Protocol		Writes to memory on a bus.
Mem.Write	PCI I/O Protocol		Allows BAR relative writes to PCI memory space.
Mem.Write	PCI Root Bridge I/O Protocol		Allows writes to memory mapped I/O space.
MetaiMatch	Unicode Collation Protocol		Performs a case insensitive comparison between a Unicode pattern string and a Unicode string.
Mtftp	PXE Base Code Protocol		Is used to perform TFTP and MTFTP services.
No associated function	EFI Device Path Protocol		Can be used on any device handle to obtain generic path/location information concerning the physical device or logical device.
No associated function	EFI Driver Entry Point		The main entry point for an EFI Driver.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
NVData	Simple Network Protocol		Allows read and writes to the NVRAM device attached to a network interface.
NvData	UNDI Commands		This command is used to read and write (if supported by NIC hardware) nonvolatile storage on the NIC.
Open	File System Protocol		Opens or creates a new file.
OpenProtocol	Boot Services	Protocol Handler Services	Adds elements to the list of agents consuming a protocol interface.
OpenProtocolInformation	Boot Services	Protocol Handler Services	Retrieve the list of agents that are currently consuming a protocol interface.
OpenVolume	Simple File System Protocol		Opens the volume for file I/O access.
OptionsValid	EFI Driver Configuration Protocol		Tests to see if a controller's current configuration options are valid.
OutputString	Simple Text Output Protocol		Displays the Unicode string on the device at the current cursor location.
PassThru	SCSI Passthru Protocol		Sends a SCSI Request Packet to a SCSI device that is connected to the SCSI channel.
Pci.Read	Device I/O Protocol		Reads from PCI Configuration Space.
Pci.Read	PCI I/O Protocol		Allows PCI controller relative reads to PCI configuration space.
Pci.Read	PCI Root Bridge I/O Protocol		Allows reads from PCI configuration space.
Pci.Write	Device I/O Protocol		Writes to PCI Configuration Space.
Pci.Write	PCI I/O Protocol		Allows PCI controller relative writes to PCI configuration space.
Pci.Write	PCI Root Bridge I/O Protocol		Allows writes to PCI configuration space
PciDevicePath	Device I/O Protocol		Provides an EFI Device Path for a PCI device with the given PCI configuration space address.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
Poll	Debugport Protocol		Determine if there is any data available to be read from the debugport device.
PollIo	PCI I/O Protocol		Polls an address in PCI I/O space until an exit condition is met, or a timeout occurs.
PollIo	PCI Root Bridge I/O Protocol		Polls an address in I/O space until an exit condition is met, or a timeout occurs.
PollMem	PCI I/O Protocol		Polls an address in PCI memory space until an exit condition is met, or a timeout occurs
PollMem	PCI Root Bridge I/O Protocol		Polls an address in memory mapped I/O space until an exit condition is met, or a timeout occurs.
ProtocolsPerHandle	Boot Services	Protocol Handler Services	Retrieves the list of protocols installed on a handle. The return buffer is automatically allocated.
QueryMode	Simple Text Output Protocol		Queries information concerning the output device's supported text mode.
RaiseTPL	Boot Services	Task Priority Services	Raises the task priority level.
Read	Debugport Protocol		Receive a buffer of characters from the debugport device.
Read	File System Protocol		Reads bytes from a file.
Read	Serial I/O Protocol		Receives a buffer of characters from a serial device.
ReadBlocks	Block I/O Protocol		Reads the requested number of blocks from the device.
ReadDisk	Disk I/O Protocol		Reads data from the disk.
ReadKeyStroke	Simple Input Protocol		Reads a keystroke from a simple input device.
Receive	Simple Network Protocol		Receives a packet from the network interface.
Receive	UNDI Commands		When the network adapter has received a frame, this command is used to copy the frame into driver/application storage.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
ReceiveFilters	UNDI Commands		This command is used to read and change receive filters and, if supported, read and change the multicast MAC address filter list.
ReceiveFilters	Simple Network Protocol		Enables and disables the receive filters for the network interface and, if supported, manages the filtered multicast HW MAC address list.
RegisterCacheFlush	EFI Byte Code Protocol		Called to register a callback function that the EBC interpreter can call to flush the processor instruction cache after creating thunks.
RegisterExceptionCallback	Debug Support Protocol		Registers a callback function that will be called each time the specified processor exception occurs.
RegisterPeriodicCallback	Debug Support Protocol		Registers a callback function that will be invoked periodically and asynchronously to the execution of EFI.
RegisterProtocolNotify	Boot Services	Protocol Handler Services	Registers for protocol interface installation notifications.
ReinstallProtocolInterface	Boot Services	Protocol Handler Services	Replaces a protocol interface.
Reset	Block I/O Protocol		Resets the block device hardware.
Reset	Debugport Protocol		Resets the debugport hardware.
Reset	Serial I/O Protocol		Resets the hardware device.
Reset	Simple Input Protocol		Resets a simple input device.
Reset	Simple Network Protocol		Resets the network adapter, and reinitializes it with the parameters that were provided in the previous call to **`Initialize()`**.
Reset	Simple Pointer Protocol		Resets the pointer device hardware.
Reset	Simple Text Output Protocol		Resets the ConsoleOut device.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
Reset	UNDI Commands		This command resets the network adapter and reinitializes the UNDI with the same parameters provided in the `Initialize()` command.
Reset	USB Host Controller Protocol		Software reset of USB.
ResetChannel	SCSI Passthru Protocol		Resets the SCSI channel.
ResetSystem	Runtime Services	Miscellaneous Services	Resets the entire platform.
ResetTarget	SCSI Passthru Protocol		Resets a SCSI device that is connected to the SCSI channel.
RestoreTPL	Boot Services	Event Services	Restores/lowers the task priority level.
RunDiagnostics	EFI Driver Diagnostics Protocol		Runs diagnostics on a controller.
SetAttribute	Simple Text Output Protocol		Sets the foreground and background color of the text that is output.
SetAttributes	PCI Root Bridge I/O Protocol		Sets attributes for a resource range on a PCI root bridge.
SetAttributes	Serial I/O Protocol		Sets communication parameters for a serial device.
SetBarAttributes	PCI I/O Protocol		Sets the attributes for a range of a BAR on a PCI controller.
SetControl	Serial I/O Protocol		Sets the control bits on a serial device.
SetCursorPosition	Simple Text Output Protocol		Sets the current cursor position.
SetInfo	File System Protocol		Sets the requested file information.
SetIpFilter	PXE Base Code Protocol		Updates the IP receive filters of a network device and enables software filtering.
SetMem	Boot Services	Miscellaneous Services	Fills a buffer with a specified value.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
SetMode	Simple Text Output Protocol		Sets the current mode of the output device.
SetMode	UGA Draw Protocol		Set the video device into the specified mode and clears the output display to black.
SetOptions	EFI Driver Configuration Protocol		Allows the user to set controller specific options for a controller that a driver is currently managing.
SetPackets	PXE Base Code Protocol		Updates the contents of the cached DHCP and Discover packets.
SetParameters	PXE Base Code Protocol		Updates the parameters that affect the operation of the PXE Base Code Protocol.
SetPosition	File System Protocol		Sets the current file position.
SetRootHubPortFeature	USB Host Controller Protocol		Sets the feature for the specified root hub port.
SetState	USB Host Controller Protocol		Sets the USB host controller to a specific state.
SetStationIp	PXE Base Code Protocol		Updates the station IP address and/or subnet mask values.
SetTime	Runtime Services	Time Services	Sets the current local time and date information.
SetTimer	Boot Services	Event Services	Sets an event to be signaled at a particular time.
SetVariable	Runtime Services	Variable Services	Sets the value of the specified variable.
SetVirtualAddressMap	Runtime Services	Virtual Memory Services	Used by an OS loader to convert from physical addressing to virtual addressing.
SetWakeupTime	Runtime Services	Time Services	Sets the system wakeup alarm clock time.
SetWatchdogTimer	Boot Services	Miscellaneous Services	Resets and sets the system's watchdog timer.
Shutdown	Boot Integrity Services Protocol		Ends the lifetime of an application instance of the `EFI_BIS` protocol, invalidating its application instance handle.
Shutdown	Simple Network Protocol		Resets the network adapter and leaves it in a state safe for another driver to initialize.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
Shutdown	UNDI Commands		Resets the network adapter and leaves it in a safe state for another driver to initialize.
SignalEvent	Boot Services	Event Services	Signals an event.
Stall	Boot Services	Miscellaneous Services	Stalls the processor.
Start	EFI Driver Binding Protocol		Starts a device controller or a bus controller.
Start	PXE Base Code Protocol		Enables the use of PXE Base Code Protocol functions.
Start	Simple Network Protocol		Changes the network interface from the stopped state to the started state.
Start	UNDI Commands		This command is used to change the UNDI operational state from stopped to started.
StartImage	Boot Services	Image Services	Function to transfer control to the Image's entry point.
Station Address	UNDI Commands		This command is used to get current station and broadcast MAC addresses and, if supported, to change the current station MAC address.
StationAddress	Simple Network Protocol		Allows the station address of the network interface to be modified.
Statistics	Simple Network Protocol		Allows the statistics on the network interface to be reset and/or collected.
Statistics	UNDI Commands		This command is used to read and clear the NIC traffic statistics.
Stop	EFI Driver Binding Protocol		Stops a device controller or a bus controller.
Stop	PXE Base Code Protocol		Disables the use of PXE Base Code Protocol functions.
Stop	Simple Network Protocol		Changes the network interface from the started state to the stopped state.
Stop	UNDI Commands		This command is used to change the UNDI operational state from started to stopped.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
StriColl	Unicode Collation Protocol		Performs a case-insensitive comparison between two Unicode strings.
StrLwr	Unicode Collation Protocol		Converts all the Unicode characters in a Null-terminated Unicode string to lower case Unicode characters.
StrToFat	Unicode Collation Protocol		Converts a Null-terminated Unicode string to legal characters in a FAT filename using an OEM character set.
StrUpr	Unicode Collation Protocol		Converts all the Unicode characters in a Null-terminated Unicode string to upper case Unicode characters.
Supported	EFI Driver Binding Protocol		Tests to see if driver supports a given controller, and further tests to see if driver supports creating a handle for a specified child device.
SyncInterruptTransfer	USB Host Controller Protocol		Submits a synchronous interrupt transfer to an interrupt endpoint of a USB device.
TestString	Simple Text Output Protocol		Tests to see if the ConsoleOut device supports this Unicode string.
Transmit	Simple Network Protocol		Places a packet in the transmit queue of the network interface.
Transmit	UNDI Commands		The Transmit command is used to place a packet into the transmit queue.
UdpRead	PXE Base Code Protocol		Reads a UDP packet from a network interface.
UdpWrite	PXE Base Code Protocol		Writes a UDP packet to a network interface.
UninstallMultipleProtocolInterfaces	Boot Services	Protocol Handler Services	Uninstalls one or more protocol interfaces from a handle.
UninstallProtocolInterface	Boot Services	Protocol Handler Services	Removes a protocol interface from a device handle.
Unload	Loaded Image Protocol		Requests an image to unload.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
UnloadImage	Boot Services	Image Services	Unloads an image.
UnloadImage	EFI Byte Code Protocol		Called when an EBC image is unloaded to allow the interpreter to perform any cleanup associated with the image's execution.
Unmap	Device I/O Protocol		Releases any resources allocated by `Map()`.
Unmap	PCI I/O Protocol		Releases any resources allocated by `Map()`.
Unmap	PCI Root Bridge I/O Protocol		Releases any resources allocated by `Map()`.
UpdateBootObject Authorization	Boot Integrity Services Protocol		Requests that the configuration parameters be altered by installing or removing an authorization certificate or changing the setting of the check flag.
UsbAsyncInterruptTransfer	USB I/O Protocol		Nonblock USB interrupt transfer.
UsbAsyncIsochronous Transfer	USB I/O Protocol		Nonblock USB isochronous transfer.
UsbBulkTransfer	USB I/O Protocol		Accesses the USB Device through USB Bulk Transfer Pipe.
UsbControlTransfer	USB I/O Protocol		Accesses the USB Device through USB Control Transfer Pipe.
UsbGetConfigDescriptor	USB I/O Protocol		Retrieves the activated configuration descriptor of a USB device.
UsbGetDeviceDescriptor	USB I/O Protocol		Retrieves the device descriptor of a USB device.
UsbGetEndpointDescriptor	USB I/O Protocol		Retrieves the endpoint descriptor of a USB Controller.
UsbGetInterfaceDescriptor	USB I/O Protocol		Retrieves the interface descriptor of a USB Controller.
UsbGetStringDescriptor	USB I/O Protocol		Retrieves the string descriptor inside a USB Device.
UsbGetSupported Languages	USB I/O Protocol		Retrieves the array of languages that the USB device supports.
UsbIsochronousTransfer	USB I/O Protocol		Accesses the USB Device through USB Isochronous Transfer Pipe.

continued

Table K-1. Functions Listed in Alphabetic Order (continued)

Function Name	Service or Protocol	Subservice	Function Description
UsbPortReset	USB I/O Protocol		Resets and reconfigures the USB controller.
UsbSyncInterruptTransfer	USB I/O Protocol		Accesses the USB Device through USB Synchronous Interrupt Transfer Pipe.
VerifyBootObject	Boot Integrity Services Protocol		Verifies a boot object according to the supplied digital signature and the current authorization certificate and check flag setting.
VerifyObjectWithCredential	Boot Integrity Services Protocol		Verifies a data object according to a supplied digital signature and a supplied digital certificate.
WaitForEvent	Boot Services	Event Services	Stops execution until an event is signaled.
Write	Debugport Protocol		Send a buffer of characters to the debugport device.
Write	File System Protocol		Writes bytes to a file.
Write	Serial I/O Protocol		Sends a buffer of characters to a serial device.
WriteBlocks	Block I/O Protocol		Writes the requested number of blocks to the device.
WriteDisk	Disk I/O Protocol		Writes data to the disk.

Table K-2. Functions Listed Alphabetically within a Service or Protocol

Service or Protocol	Function	Function Description
Block I/O Protocol	FlushBlocks	Flushes any cached blocks.
	ReadBlocks	Reads the requested number of blocks from the device.
	Reset	Resets the block device hardware.
	WriteBlocks	Writes the requested number of blocks to the device.
Boot Integrity Services Protocol	Free	Frees memory structures allocated and returned by other functions in the **EFI_BIS** protocol.
	GetBootObjectAuthorization Certificate	Retrieves the current digital certificate (if any) used by the **EFI_BIS** protocol as the source of authorization for verifying boot objects and altering configuration parameters
	GetBootObjectAuthorization CheckFlag	Retrieves the current setting of the authorization check flag that indicates whether or not authorization checks are required for boot objects.
	GetBootObjectAuthorization UpdateToken	Retrieves an uninterpreted token whose value gets included and signed in a subsequent request to alter the configuration parameters, to protect against attempts to “replay” such a request.
	GetSignatureInfo	Retrieves information about the digital signature algorithms supported and the identity of the installed authorization certificate, if any.
	Initialize	Initializes an application instance of the **EFI_BIS** protocol, returning a handle for the application instance.
	Shutdown	Ends the lifetime of an application instance of the **EFI_BIS** protocol, invalidating its application instance handle.
	UpdateBootObject Authorization	Requests that the configuration parameters be altered by installing or removing an authorization certificate or changing the setting of the check flag.
	VerifyBootObject	Verifies a boot object according to the supplied digital signature and the current authorization certificate and check flag setting.
	VerifyObjectWithCredential	Verifies a data object according to a supplied digital signature and a supplied digital certificate.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
Boot Services	AllocatePages	Allocates memory pages of a particular type.
	AllocatePool	Allocates pool of a particular type.
	CalculateCrc32	Computes and returns a 32-bit CRC for a data buffer.
	CheckEvent	Checks whether an event is in the signaled state.
	CloseEvent	Closes and frees an event structure.
	CloseProtocol	Removes elements from the list of agents consuming a protocol interface.
	ConnectController	Uses a set of precedence rules to find the best set of drivers to manage a controller.
	CopyMem	Copies the contents of one buffer to another buffer.
	CreateEvent	Creates a general-purpose event structure.
	DisconnectController	Informs a set of drivers to stop managing a controller.
	EFI_IMAGE_ ENTRY_POINT	Prototype of an EFI Image's entry point.
	Exit	Exits the image's entry point.
	ExitBootServices	Terminates boot services.
	FreePages	Frees memory pages.
	FreePool	Frees allocated pool.
	GetMemoryMap	Returns the current boot services memory map and memory map key.
	GetNextMonotonicCount	Returns a monotonically increasing count for the platform.
	HandleProtocol	Queries the list of protocol handlers on a device handle for the requested Protocol Interface.
	InstallConfigurationTable	Adds, updates, or removes a configuration table from the EFI System Table
	InstallMultipleProtocol Interfaces	Installs one or more protocol interfaces onto a handle.
	InstallProtocolInterface	Adds a protocol interface to an existing or new device handle.
	LoadImage	Function to dynamically load another EFI Image.
	LocateDevicePath	Locates the closest handle that supports the specified protocol on the specified device path.
	LocateHandle	Locates the handle(s) that support the specified protocol.
	LocateHandleBuffer	Retrieves the list of handles from the handle database that meet the search criteria. The return buffer is automatically allocated.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
Boot Services (continued)	LocateProtocol	Finds the first handle in the handle database the supports the requested protocol.
	OpenProtocol	Adds elements to the list of agents consuming a protocol interface.
	OpenProtocolInformation	Retrieve the list of agents that are currently consuming a protocol interface.
	ProtocolsPerHandle	Retrieves the list of protocols installed on a handle. The return buffer is automatically allocated.
	RaiseTPL	Raises the task priority level.
	RegisterProtocolNotify	Registers for protocol interface installation notifications
	ReinstallProtocolInterface	Replaces a protocol interface.
	RestoreTPL	Restores/lowers the task priority level.
	SetMem	Fills a buffer with a specified value.
	SetTimer	Sets an event to be signaled at a particular time.
	SetWatchdogTimer	Resets and sets the system's watchdog timer.
	SignalEvent	Signals an event.
	Stall	Stalls the processor.
	StartImage	Function to transfer control to the Image's entry point.
	UninstallMultipleProtocol Interfaces	Uninstalls one or more protocol interfaces from a handle.
	UninstallProtocolInterface	Removes a protocol interface from a device handle.
	UnloadImage	Unloads an image.
	WaitForEvent	Stops execution until an event is signaled.
Debugport Protocol	Poll	Determine if there is any data available to be read from the debugport device.
	Read	Receive a buffer of characters from the debugport device.
	Reset	Resets the debugport hardware.
	Write	Send a buffer of characters to the debugport device.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
Debug Support Protocol	GetMaximumProcessor Index	Returns the maximum processor index value that may be used with `RegisterPeriodicCallback()` and `RegisterExceptionCallback()`.
	InvalidateInstructionCache	Invalidate the instruction cache of the processor.
	RegisterExceptionCallback	Registers a callback function that will be called each time the specified processor exception occurs.
	RegisterPeriodicCallback	Registers a callback function that will be invoked periodically and asynchronously to the execution of EFI.
Decompress Protocol	Decompress	Decompresses a compressed source buffer into an uncompressed destination buffer.
	GetInfo	Given the compressed source buffer, this function retrieves the size of the uncompressed destination buffer and the size of the scratch buffer required to perform the decompression.
Device I/O Protocol	AllocateBuffer	Allocates pages that are suitable for a common buffer mapping.
	Flush	Flushes any posted write data to the device.
	FreeBuffer	Frees pages that were allocated with `AllocateBuffer()`.
	Io.Read	Reads from I/O ports on a bus.
	Io.Write	Writes to I/O ports on a bus.
	Map	Provides the device specific addresses needed to access host memory for DMA.
	Mem.Read	Reads from memory on a bus.
	Mem.Write	Writes to memory on a bus.
	Pci.Read	Reads from PCI Configuration Space.
	Pci.Write	Writes to PCI Configuration Space.
	PciDevicePath	Provides an EFI Device Path for a PCI device with the given PCI configuration space address.
	Unmap	Releases any resources allocated by `Map()`.
Disk I/O Protocol	ReadDisk	Reads data from the disk.
	WriteDisk	Writes data to the disk.
EFI Bus-Specific Driver Override Protocol	GetDriver	Uses a bus specific algorithm to retrieve a driver image handle for a controller.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
EFI Byte Code Protocol	CreateThunk	Creates a thunk for an EBC image entry point or protocol service, and returns a pointer to the thunk.
	RegisterCacheFlush	Called to register a callback function that the EBC interpreter can call to flush the processor instruction cache after creating thunks.
	UnloadImage	Called when an EBC image is unloaded to allow the interpreter to perform any cleanup associated with the image's execution.
EFI Component Name Protocol	GetControllerName	Retrieves a Unicode string that is the user readable name of the controller that is being managed by an EFI Driver.
	GetDriverName	Retrieves a Unicode string that is the user readable name of the EFI Driver.
EFI Device Path Protocol	No associated function	Can be used on any device handle to obtain generic path/location information concerning the physical device or logical device.
EFI Driver Binding Protocol	Start	Starts a device controller or a bus controller.
	Stop	Stops a device controller or a bus controller.
	Supported	Tests to see if driver supports a given controller, and further tests to see if driver supports creating a handle for a specified child device.
EFI Driver Configuration Protocol	ForceDefaults	Forces a driver to set the default configuration options for a controller.
	OptionsValid	Tests to see if a controller's current configuration options are valid.
	SetOptions	Allows the user to set controller specific options for a controller that a driver is currently managing.
EFI Driver Diagnostics Protocol	RunDiagnostics	Runs diagnostics on a controller.
EFI Driver Entry Point	No associated function	The main entry point for an EFI Driver.
EFI Driver Override Protocol	DriverLoaded	Used to associate a driver image handle with a device path returned on a prior call.
	GetDriver	Retrieves the image handle of the platform override driver for a controller in the system.
	GetDriverPath	Retrieves the device path of the platform override driver for a controller in the system.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
File System Protocol	Close	Closes the current file handle.
	Delete	Deletes a file.
	Flush	Flushes all modified data associated with the file to the device.
	GetInfo	Gets the requested file or volume information.
	GetPosition	Returns the current file position.
	Open	Opens or creates a new file.
	Read	Reads bytes from a file.
	SetInfo	Sets the requested file information.
	SetPosition	Sets the current file position.
	Write	Writes bytes to a file.
Load File Protocol	LoadFile	Causes the driver to load the requested file.
Loaded Image Protocol	Unload	Requests an image to unload.
PCI I/O Protocol	AllocateBuffer	Allocates pages that are suitable for a common buffer mapping.
	Attributes	Performs an operation on the attributes that this PCI controller supports.
	CopyMem	Allows one region of PCI memory space to be copied to another region of PCI memory space
	Flush	Flushes all PCI posted write transactions to system memory.
	FreeBuffer	Frees pages that were allocated with `AllocateBuffer()`.
	GetBarAttributes	Gets the attributes that this PCI controller supports setting on a BAR using `SetBarAttributes()`, and retrieves the list of resource descriptors for a BAR.
	GetLocation	Retrieves this PCI controller's current PCI bus number, device number, and function number.
	Io.Read	Allows BAR relative reads to PCI I/O space.
	Io.Write	Allows BAR relative writes to PCI I/O space.
	Map	Provides the PCI controller specific address needed to access system memory for DMA.
	Mem.Read	Allows BAR relative reads to PCI memory space.
	Mem.Write	Allows BAR relative writes to PCI memory space.
	Pci.Read	Allows PCI controller relative reads to PCI configuration space.
	Pci.Write	Allows PCI controller relative writes to PCI configuration space.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
PCI I/O Protocol (continued)	PollIo	Polls an address in PCI I/O space until an exit condition is met, or a timeout occurs.
	PollMem	Polls an address in PCI memory space until an exit condition is met, or a timeout occurs
	SetBarAttributes	Sets the attributes for a range of a BAR on a PCI controller.
	Unmap	Releases any resources allocated by `Map()`.
PCI Root Bridge I/O Protocol	AllocateBuffer	Allocates pages that are suitable for a common buffer mapping.
	Configuration	Gets the current resource settings for this PCI root bridge
	CopyMem	Allows one region of PCI root bridge memory space to be copied to another region of PCI root bridge memory space.
	Flush	Flushes all PCI posted write transactions to system memory.
	FreeBuffer	Free pages that were allocated with `AllocateBuffer()`.
	GetAttributes	Gets the attributes that a PCI root bridge supports setting with `SetAttributes()`, and the attributes that a PCI root bridge is currently using.
	Io.Read	Allows reads from I/O space.
	Io.Write	Allows writes to I/O space.
	Map	Provides the PCI controller specific addresses needed to access system memory for DMA.
	Mem.Read	Allows reads from memory mapped I/O space.
	Mem.Write	Allows writes to memory mapped I/O space.
	Pci.Read	Allows reads from PCI configuration space.
	Pci.Write	Allows writes to PCI configuration space
	PollIo	Polls an address in I/O space until an exit condition is met, or a timeout occurs.
	PollMem	Polls an address in memory mapped I/O space until an exit condition is met, or a timeout occurs.
	SetAttributes	Sets attributes for a resource range on a PCI root bridge.
	Unmap	Releases any resources allocated by `Map()`.
PXE Base Code Callback Protocol	Callback	Callback routine used by the PXE Base Code `Dhcp()`, `Discover()`, `Mtftp()`, `UdpWrite()`, and `Arp()` functions.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
PXE Base Code Protocol	Arp	Uses the ARP protocol to resolve a MAC address.
	Dhcp	Attempts to complete a DHCPv4 D.O.R.A. (discover / offer / request / acknowledge) or DHCPv6 S.A.R.R (solicit / advertise / request / reply) sequence.
	Discover	Attempts to complete the PXE Boot Server and/or boot image discovery sequence.
	EFI_PXE_BASE_CODE _CALLBACK	Callback function that is invoked when the PXE Base Code Protocol is waiting for an event.
	Mtftp	Is used to perform TFTP and MTFTP services.
	SetIpFilter	Updates the IP receive filters of a network device and enables software filtering.
	SetPackets	Updates the contents of the cached DHCP and Discover packets.
	SetParameters	Updates the parameters that affect the operation of the PXE Base Code Protocol.
	SetStationIp	Updates the station IP address and/or subnet mask values.
	Start	Enables the use of PXE Base Code Protocol functions.
	Stop	Disables the use of PXE Base Code Protocol functions.
	UdpRead	Reads a UDP packet from a network interface.
	UdpWrite	Writes a UDP packet to a network interface.
Runtime Services	ConvertPointer	Used by EFI components to convert internal pointers when switching to virtual addressing.
	GetNextHigh MonotonicCount	Returns the next high 32 bits of a platform's monotonic counter.
	GetNextVariableName	Enumerates the current variable names.
	GetTime	Returns the current time and date, and the time-keeping capabilities of the platform.
	GetVariable	Returns the value of the specific variable.
	GetWakeupTime	Returns the current wakeup alarm clock setting.
	ResetSystem	Resets the entire platform.
	SetTime	Sets the current local time and date information.
	SetVariable	Sets the value of the specified variable.
	SetVirtualAddressMap	Used by an OS loader to convert from physical addressing to virtual addressing.
	SetWakeupTime	Sets the system wakeup alarm clock time.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
SCSI Passthru Protocol	BuildDevicePath	Used to allocate and build a device path node for a SCSI device on a SCSI channel.
	GetNextDevice	Used to retrieve the list of legal Target IDs for the SCSI devices on a SCSI channel.
	GetTargetLun	Used to translate a device path node to a Target ID and LUN.
	PassThru	Sends a SCSI Request Packet to a SCSI device that is connected to the SCSI channel.
	ResetChannel	Resets the SCSI channel.
	ResetTarget	Resets a SCSI device that is connected to the SCSI channel.
Serial I/O Protocol	GetControl	Reads the status of the control bits on a serial device.
	Read	Receives a buffer of characters from a serial device.
	Reset	Resets the hardware device.
	SetAttributes	Sets communication parameters for a serial device.
	SetControl	Sets the control bits on a serial device.
	Write	Sends a buffer of characters to a serial device.
Simple File System Protocol	OpenVolume	Opens the volume for file I/O access.
Simple Input Protocol	ReadKeyStroke	Reads a keystroke from a simple input device.
	Reset	Resets a simple input device.
Simple Network Protocol	GetStatus	Reads the current interrupt status and recycled transmit buffer status from the network interface.
	Initialize	Resets the network adapter and allocates the transmit and receive buffers required by the network interface; also optionally allows space for additional transmit and receive buffers to be allocated
	MCastIPtoMAC	Allows a multicast IP address to be mapped to a multicast HW MAC address.
	NVData	Allows read and writes to the NVRAM device attached to a network interface.
	Receive	Receives a packet from the network interface.
	ReceiveFilters	Enables and disables the receive filters for the network interface and, if supported, manages the filtered multicast HW MAC address list
	Reset	Resets the network adapter, and reinitializes it with the parameters that were provided in the previous call to `Initialize()`.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
Simple Network Protocol (continued)	Shutdown	Resets the network adapter and leaves it in a state safe for another driver to initialize.
	Start	Changes the network interface from the stopped state to the started state.
	StationAddress	Allows the station address of the network interface to be modified.
	Statistics	Allows the statistics on the network interface to be reset and/or collected.
	Stop	Changes the network interface from the started state to the stopped state.
	Transmit	Places a packet in the transmit queue of the network interface.
Simple Pointer Protocol	GetState	Retrieves the current state of a pointer device.
	Reset	Resets the pointer device hardware.
Simple Text Output Protocol	ClearScreen	Clears the screen with the currently set background color.
	EnableCursor	Turns the visibility of the cursor on/off.
	OutputString	Displays the Unicode string on the device at the current cursor location.
	QueryMode	Queries information concerning the output device's supported text mode.
	Reset	Resets the ConsoleOut device.
	SetAttribute	Sets the foreground and background color of the text that is output.
	SetCursorPosition	Sets the current cursor position.
	SetMode	Sets the current mode of the output device.
	TestString	Tests to see if the ConsoleOut device supports this Unicode string.
UGA Draw Protocol	Blt	Blt a rectangle of pixels on the graphics screen. Blt stands for BLock Transfer.
	GetMode	Return the current frame buffer geometry and display refresh rate.
	SetMode	Sets the video device into the specified mode and clears the output display to black.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
UGA I/O Protocol	CreateDevice	Dynamically allocates storage for a child `UGA_DEVICE`.
	DeleteDevice	Frees a dynamically allocated child `UGA_DEVICE` object that was allocated via `CreateDevice()`.
	DispatchService	This is the main UGA service dispatch routine for all `UGA_IO_REQUEST`s.
UNDI Commands	Fill Header	This command is used to fill the media header(s) in transmit packet(s).
	Get Config Info	This command is used to retrieve configuration information about the NIC being controlled by the UNDI.
	Get Init Info	This command is used to retrieve initialization information that is needed by drivers and applications to initialized UNDI.
	Get State	This command is used to determine the operational state of the UNDI.
	Get Status	This command returns the current interrupt status and/or the transmitted buffer addresses.
	Initialize	This command resets the network adapter and initializes UNDI using the parameters supplied in the CPB.
	Interrupt Enables	The Interrupt Enables command can be used to read and/or change the current external interrupt enable settings.
	MCast IP to MAC	Translate a multicast IPv4 or IPv6 address to a multicast MAC address.
	NvData	This command is used to read and write (if supported by NIC H/W) nonvolatile storage on the NIC.
	Receive	When the network adapter has received a frame, this command is used to copy the frame into driver/application storage.
	Receive Filters	This command is used to read and change receive filters and, if supported, read and change the multicast MAC address filter list.
	Reset	This command resets the network adapter and reinitializes the UNDI with the same parameters provided in the Initialize command.
	Shutdown	The Shutdown command resets the network adapter and leaves it in a safe state for another driver to initialize.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
UNDI Commands (continued)	Start	This command is used to change the UNDI operational state from stopped to started.
	Station Address	This command is used to get current station and broadcast MAC addresses and, if supported, to change the current station MAC address.
	Statistics	This command is used to read and clear the NIC traffic statistics.
	Stop	This command is used to change the UNDI operational state from started to stopped.
	Transmit	The Transmit command is used to place a packet into the transmit queue.
Unicode Collation Protocol	FatToStr	Converts an 8.3 FAT file name in an OEM character set to a Null-terminated Unicode string.
	MetaiMatch	Performs a case insensitive comparison between a Unicode pattern string and a Unicode string.
	StriColl	Performs a case-insensitive comparison between two Unicode strings.
	StrLwr	Converts all the Unicode characters in a Null-terminated Unicode string to lower case Unicode characters.
	StrToFat	Converts a Null-terminated Unicode string to legal characters in a FAT filename using an OEM character set.
	StrUpr	Converts all the Unicode characters in a Null-terminated Unicode string to upper case Unicode characters.
USB Host Controller Protocol	AsyncInterruptTransfer	Submits an asynchronous interrupt transfer to an interrupt endpoint of a USB device.
	AsyncIsochronousTransfer	Submits nonblocking USB isochronous transfer.
	BulkTransfer	Submits a bulk transfer to a bulk endpoint of a USB device.
	ClearRootHubPortFeature	Clears the feature for the specified root hub port.
	ControlTransfer	Submits a control transfer to a target USB device.
	GetRootHubPortNumber	Retrieves the number of root hub ports that are produced by the USB host controller.
	GetRootHubPortStatus	Retrieves the status of the specified root hub port.
	GetState	Retrieves the current state of the USB host controller.

continued

Table K-2. Functions Listed Alphabetically within a Service or Protocol (continued)

Service or Protocol	Function	Function Description
USB Host Controller Protocol (continued)	IsochronousTransfer	Submits isochronous transfer to an isochronous endpoint of a USB device.
	Reset	Software reset of USB.
	SetRootHubPortFeature	Sets the feature for the specified root hub port.
	SetState	Sets the USB host controller to a specific state.
	SyncInterruptTransfer	Submits a synchronous interrupt transfer to an interrupt endpoint of a USB device.
USB I/O Protocol	UsbAsyncInterruptTransfer	Nonblock USB interrupt transfer.
	UsbAsyncIsochronous Transfer	Nonblock USB isochronous transfer.
	UsbBulkTransfer	Accesses the USB Device through USB Bulk Transfer Pipe.
	UsbControlTransfer	Accesses the USB Device through USB Control Transfer Pipe.
	UsbGetConfigDescriptor	Retrieves the activated configuration descriptor of a USB device.
	UsbGetDeviceDescriptor	Retrieves the device descriptor of a USB device.
	UsbGetEndpointDescriptor	Retrieves the endpoint descriptor of a USB Controller.
	UsbGetInterfaceDescriptor	Retrieves the interface descriptor of a USB Controller.
	UsbGetStringDescriptor	Retrieves the string descriptor inside a USB Device.
	UsbGetSupported Languages	Retrieves the array of languages that the USB device supports.
	UsbIsochronousTransfer	Accesses the USB Device through USB Isochronous Transfer Pipe.
	UsbPortReset	Resets and reconfigures the USB controller.
	UsbSyncInterruptTransfer	Accesses the USB Device through USB Synchronous Interrupt Transfer Pipe.

References

Related Information

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

- [BASE64] *RFC 1521: MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. Section 5.2: Base64 Content-Transfer-Encoding. ftp://ftp.isi.edu/in-notes/rfc1521.txt
- [PKCS] *The Public-Key Cryptography Standards*, RSA Laboratories, Redwood City, CA: RSA Data Security, Inc.
- [RFC 1700] J. Reynolds, J. Postel: Assigned Numbers | ISI, October 1994
- [RFC 2460] *Internet Protocol, Version 6 (IPv6) Specificatio*n, http://www.faqs.org/rfcs/rfc2460.html
- [RFC 791] *Internet Protocol DARPA Internet Program Protocol (IPv4) Specification*, September 1981, http://www.faqs.org/rfcs/rfc791.html
- [SM spec] *Common Security: CDSA and CSSM*, Version 2 (with corrigenda), was *Signed Manifest Specification*, The Open Group, May 2000. http://www.opengroup.org/pubs/catalog/c914.htm
- *"El Torito" Bootable CD-ROM Format Specification,* Version 1.0, Phoenix Technologies, Ltd., IBM Corporation, 1994, http://www.phoenix.com/en/support/white+papers-specs/
- *Advanced Configuration and Power Interface Specification*, Intel, Microsoft, Toshiba, Compaq, and Phoenix, Revision 2.0, July 27, 2000, http://acpi.info/index.html
- **Address Resolution Protocol** – http://www.ietf.org/rfc/rfc0826.txt. Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- *Advanced Configuration and Power Interface Specification*, Revision 2.0, July 27, 2000, http://www.acpi.info/spec.htm
- **Assigned Numbers** – Lists the reserved numbers used in the RFCs and in this specification - http://www.ietf.org/rfc/rfc1700.txt. Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- *BIOS Boot Specification Version 1.01*, Compaq Computer Corporation, Phoenix Technologies Ltd., Intel Corporation, 1996, http://www.phoenix.com/en/support/white+papers-specs/
- **Bootstrap Protocol** – http://www.ietf.org/rfc/rfc0951.txt - This reference is included for backward compatibility. BC protocol supports DHCP and BOOTP. Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- *CAE Specification [UUID], DCE 1.1:Remote Procedure Call,* Document Number C706, *Universal Unique Identifier Appendix,* Copyright © 1997, The Open Group, http://www.opengroup.org/onlinepubs/9629399/toc.htm
- *Clarification to Plug and Play BIOS Specification Version 1.0*, http://www.microsoft.com/hwdev/tech/pnp/

- **Dynamic Host Configuration Protocol –** DHCP for Ipv4 (protocol: http://www.ietf.org/rfc/rfc2131.txt, options: http://www.ietf.org/rfc/rfc2132.txt). Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- *EFI Specification Version 1.02*, Intel Corporation, 2000, http://developer.intel.com/technology/efi.
- *File Verification Using CRC*, Mark R. Nelson, Dr. Dobbs, May 1994
- *Hardware Design Guide Version 3.0 for Microsoft Windows 2000 Server*, Intel Corporation, Microsoft Corporation, 2000, http://developer.intel.com/design/servers/desguide/hdgv3.htm
- *IA-32 Intel Architecture Software Developer's Manual*, Intel Corporation, 2001, http://www.intel.com/design/pentium4/manuals/
- *Information Technology — BIOS Enhanced Disk Drive Services (EDD*), working draft T13/1386D, Revision 5a, September 28, 2000, http://t13.org/project/d1386r5a.pdf
- *Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture, Rev. 1.0*, Order number 245317, Intel Corporation, January, 2000. Also available at http://developer.intel.com/design/itanium/family/
- *Itanium® Architecture Software Developer's Manual, Volume 2: System Architecture, Rev. 1.0*, Order number 245318, Intel Corporation, January, 2000. Also available at http://developer.intel.com/design/itanium/family/
- *Itanium® Architecture Software Developer's Manual, Volume 3: Instruction Set Reference, Rev. 1.0*, Order number 245319, Intel Corporation, January, 2000. Also available at http://developer.intel.com/design/itanium/family/
- *Itanium® Architecture Software Developer's Manual, Volume 4: Itanium Processor Programmer's Guide, Rev. 1.0*, Order number 245320, Intel Corporation, January 2000. Also available at http://developer.intel.com/design/itanium/family/
- *Itanium® Software Conventions and Runtime Architecture Guide,* Order number 245358, Intel Corporation, January, 2000. Also available at http://developer.intel.com/design/itanium/family/
- *Itanium® System Abstraction Layer Specification,* Available at http://developer.intel.com/design/itanium/family/
- *IEEE 1394 Specification*, http://www.1394ta.org/Technology/Specifications/specifications.htm
- **Internet Control Message Protocol –** ICMP for Ipv4: http://www.ietf.org/rfc/rfc0792.txt. ICMP for Ipv6: http://www.ietf.org/rfc/rfc2463.txt. Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- **Internet Engineering Task Force** – http://www.ietf.org/. Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- **Internet Group Management Protocol** – http://www.ietf.org/rfc/rfc2236.txt . Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- **Internet Protocol -** Ipv4: http://www.ietf.org/rfc/rfc0791.txt. Ipv6: http://www.ietf.org/rfc/rfc2460.txt & http://www.ipv6.org. Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.

- *ISO 639-2:1998.* Codes for the Representation of Names of Languages – Part2: Alpha-3 code, http://www.iso.ch/
- ISO/IEC 3309:1991(E), *Information Technology - Telecommunications and information exchange between systems - High-level data link control (HDLC) procedures - Frame structure*, International Organization For Standardization, Fourth edition 1991-06-01
- ITU-T Rec. V.42, *Error-Correcting Procedures for DCEs using asynchronous-to-synchronous conversion*, October, 1996
- *Microsoft Extensible Firmware Initiative FAT32 File System Specification*, Version 1.03, Microsoft Corporation, December 6, 2000, http://www.microsoft.com/hwdev/specs/
- *Microsoft Portable Executable and Common Object File Format Specification,* Version 6.0, http://www.microsoft.com/hwdev/specs/, Microsoft Corporation, May 25, 2000
- *OSTA Universal Disk Format Specification*, Revision 2.00, Optical Storage Technology Association, 1998, http://www.osta.org/specs/
- *PCI BIOS Specification,* Revision 2.1, PCI Special Interest Group, Hillsboro, OR, http://www.pcisig.com/specifications
- *PCI Hot-Plug Specification* Revision 1.0, PCI Special Interest Group, Hillsboro, OR, http://www.pcisig.com/specifications
- *PCI Local Bus Specification* Revision 2.2, PCI Special Interest Group, Hillsboro, OR, http://www.pcisig.com/specifications
- *Plug and Play BIOS Specification*, Version 1.0A, Compaq Computer Corporation, Phoenix Technologies, Ltd., Intel Corporation, 1994, http://www.microsoft.com/hwdev/tech/pnp/
- **Plug and Play** – http://www.phoenix.com/en/support/white+papers-specs/ Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- *Portable Executable and Common Object File Format Specification*. See http://www.microsoft.com/hwdev/hardware/PECOFF.asp
- *POST Memory Manager Specification,* Version 1.01, Phoenix Technologies Ltd., Intel Corporation, 1997, http://www.phoenix.com/en/support/white+papers-specs/
- *Preboot Execution Environment (PXE) Specification*, Version 2.1. Intel Corporation, 1999. Available at ftp://download.intel.com/labs/manage/wfm/download/pxespec.pdf.
- **Request For Comments** – http://www.ietf.org/rfc.html and http://www.keywave.ad.jp/RFC/index.html. Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- *SYSID BIOS Support Interface Requirements,* Version 1.2, Intel Corporation, 1997, http://www.intel.com/labs/manage/wfm/wfmspecs.htm
- *SYSID Programming Interface* Version 1.2, http://www.intel.com/labs/manage/wfm/wfmspecs.htm
- *System Management BIOS Reference Specification*, Version 2.3, American Megatrends Inc., Award Software International Inc., Compaq Computer Corporation, Dell Computer Corporation, Hewlett-Packard Company, Intel Corporation, International Business Machines Corporation, Phoenix Technologies Limited, and SystemSoft Corporation, 1977, 1998, http://www.dmtf.org/standards/bios.php or http://www.phoenix.com/en/support/white+papers-specs/
- **Transmission Control Protocol** – TCPv4: http://www.ietf.org/rfc/rfc0793.txt. TCPv6: ftp://ftp.ipv6.org/pub/rfc/rfc2147.txt. Refer to Appendix E,"32/64-Bit UNDI Specification," for more information.

- **Trivial File Transfer Protocol** – TFTP (protocol: http://www.ietf.org/rfc/rfc1350.txt, options: http://www.ietf.org/rfc/rfc2347.txt, http://www.ietf.org/rfc/rfc2348.txt and http://www.ietf.org/rfc/rfc2349.txt). Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- **User Datagram Protocol –** UDP over IPv4: http://www.ietf.org/rfc/rfc0768.txt. UDP over IPv6: http://www.ietf.org/rfc/rfc2454.txt. Refer to Appendix E, "32/64-Bit UNDI Specification," for more information.
- *The Unicode Standard,* Version 2.1, Unicode Consortium, http://www.unicode.org/
- More information on EFI 1.10 UGA ROM usage under an OS can be found at www.microsoft.com/hwdev/uga.
- *Universal Serial Bus PC Legacy Compatibility Specification*, Version 0.9, http://www.usb.org/developers/docs.html
- *Wired for Management Baseline,* Version 2.0 Release Candidate. Intel Corporation, 1998, http://www.intel.com/labs/manage/wfm/wfmspecs.htm

Prerequisite Specifications

In general, this specification requires that functionality defined in a number of other existing specifications be present on a system that implements this specification. This specification requires that those specifications be implemented at least to the extent that all the required elements are present.

This specification prescribes the use and extension of previously established industry specification tables whenever possible. The trend to remove runtime call-based interfaces is well documented. The ACPI (Advanced Configuration and Power Interface) specification and the SAL (System Access Layer) specification are two examples of new and innovative firmware technologies that were designed on the premise that OS developers prefer to minimize runtime calls into firmware. ACPI focuses on no runtime calls to the BIOS, and the SAL specification only supports runtime services that make the OS more portable.

ACPI Specification

The interface defined by the *Advanced Configuration and Power Interface* (ACPI) S*pecification* is the current state-of-the-art in the platform-to-OS interface. ACPI fully defines the methodology that allows the OS to discover and configure all platform resources. ACPI allows the description of non-Plug and Play motherboard devices in a plug and play manner. ACPI also is capable of describing power management and hot plug events to the OS. (For more information on ACPI, refer to the ACPI web site at http://www.acpi.info/spec.htm).

WfM Specification

The *Wired for Management (WfM) Specification* defines a baseline for manageability that can be used to lower the total cost of ownership of a computer system. WfM includes the System Management BIOS (SMBIOS) table-based interface that is used by the platform to relate platform-specific management information to the OS or an OS-based management agent. The format of the data is defined in the *System Management BIOS Reference Specification*, and it is up to higher level software to map the information provided by the platform into the appropriate schema. Examples of schema would include CIM (Common Information Model) and DMI (Desktop Management Interface). For more information on WfM or to obtain a copy of the WfM Specification, visit http://www.intel.com/labs/manage/wfm/wfmspecs.htm. To obtain the *System Management BIOS Reference Specification*, visit http://www.phoenix.com/en/support/white+papers-specs/.

Additional Considerations for Itanium-Based Platforms

Any information or service that is available in Itanium architecture firmware specifications supercedes any requirement in the common IA-32 and Itanium architecture specifications listed above. The Itanium architecture firmware specifications (currently the *Itanium® System Abstraction Layer Specification* and portions of the *Intel® Itanium® Architecture Software Developer's Manual,* volumes 1–4) define the baseline functionality required for all Itanium architecture platforms. The major addition that EFI makes to these Itanium architecture firmware specifications is that it defines a boot infrastructure and a set of services that constitute a common platform definition for high-volume Itanium architecture–based systems to implement based on the more generalized Itanium architecture firmware specifications.

The following specifications are the required Intel Itanium architecture specifications for all Itanium architecture–based platforms:

- *Itanium® Processor Family System Abstraction Layer Specification*
- *Intel® Itanium® Architecture Software Developer's Manual,* volumes 1–4

Both documents are available at http://developer.intel.com/design/itanium/family/.

Glossary

_ADR	A reserved name in ACPI name space. It refers to an address on a bus that has standard enumeration. An example would be PCI, where the enumeration method is described in the PCI Local Bus specification.
_CRS	A reserved name in ACPI name space. It refers to the current resource setting of a device. A _CRS is required for devices that are not enumerated in a standard fashion. _CRS is how ACPI converts nonstandard devices into Plug and Play devices.
_HID	A reserved name in ACPI name space. It represents a device's plug and play hardware ID and is stored as a 32-bit compressed EISA ID. _HID objects are optional in ACPI. However, a _HID object must be used to describe any device that will be enumerated by the ACPI driver in the OS. This is how ACPI deals with non–Plug and Play devices.
_UID	A reserved name in ACPI name space. It is a serial number style ID that does not change across reboots. If a system contains more than one device that reports the same _HID, each device must have a unique _UID. The _UID only needs to be unique for device that have the exact same _HID value.
ACPI Device Path	A Device Path that is used to describe devices whose enumeration is not described in an industry-standard fashion. These devices must be described using ACPI AML in the ACPI name space; this type of node provides linkage to the ACPI name space.
ACPI	Refers to the *Advanced Configuration and Power Interface Specification* and to the concepts and technology it discusses. The specification defines a new interface to the system board that enables the operating system to implement operating system-directed power management and system configuration.
Base Code (BC)	The PXE Base Code, included as a core protocol in EFI, is comprised of a simple network stack (UDP/IP) and a few common network protocols (DHCP, Bootserver Discovery, TFTP) that are useful for remote booting machines.
BC	See Base Code
Big Endian	A memory architecture in which the low-order byte of a multibyte datum is at the highest address, while the high-order byte is at the lowest address. See Little Endian.
BIOS Boot Specification Device Path	A Device Path that is used to point to boot legacy operating systems; it is based on the *BIOS Boot Specification*, Version 1.01.

BIOS Parameter Block (BPB)	The first block (sector) of a partition. It defines the type and location of the **FAT File System** on a drive.
BIOS	Basic Input/Output System. A collection of low-level I/O service routines.
Block I/O Protocol	A protocol that is used during boot services to abstract mass storage devices. It allows boot services code to perform block I/O without knowing the type of a device or its controller.
Block Size	The fundamental allocation unit for devices that support the **Block I/O Protocol**. Not less than 512 bytes. This is commonly referred to as sector size on hard disk drives.
Boot Device	The **device handle** that corresponds to the device from which the currently executing image was loaded.
Boot Manager	The part of the firmware implementation that is responsible for implementing system boot policy. Although a particular boot manager implementation is not specified in this document, such code is generally expected to be able to enumerate and handle transfers of control to the available OS loaders as well as EFI applications and drivers on a given system. The boot manager would typically be responsible for interacting with the system user, where applicable, to determine what to load during system startup. In cases where user interaction is not indicated, the boot manager would determine what to load and, if multiple items are to be loaded, what the sequencing of such loads would be.
Boot Services Driver	A program that is loaded into boot services memory and stays resident until boot services terminates.
Boot Services Table	A table that contains the firmware entry points for accessing boot services functions such as **Task Priority Services** and **Memory Allocation Services**. The table is accessed through a pointer in the **System Table**.
Boot Services Time	The period of time between platform initialization and the call to `ExitBootServices()`. During this time, **EFI drivers** and applications are loaded iteratively and the system boots from an ordered list of EFI OS loaders.
Boot Services	The collection of interfaces and protocols that are present in the boot environment. The services minimally provide an OS loader with access to platform capabilities required to complete OS boot. Services are also available to drivers and applications that need access to platform capability. Boot services are terminated once the operating system takes control of the platform.
BPB	See **BIOS Parameter Block**.

CIM See **Common Information Model**.

Cluster A collection of disk sectors. Clusters are the basic storage units for disk files. See **File Allocation Table**.

COFF Common Object File Format, a standard file format for binary images.

Coherency Domain
(1) The global set of resources that is visible to at least one processor in a platform.
(2) The address resources of a system as seen by a processor. It consists of both system memory and I/O space.

Common Information Model (CIM)
An object-oriented schema defined by the **DMTF**. CIM is an information model that provides a common way to describe and share management information enterprise-wide.

Console I/O Protocol
A protocol that is used during **boot services** to handle input and output of text-based information intended for the system administrator. It has two parts, a **Simple Input Protocol** that is used to obtain input from the **ConsoleIn** device and a **Simple Text Output Protocol** that is used to control text-based output devices. The **Console I/O Protocol** is also known as the EFI Console I/O Protocol.

ConsoleIn The device handle that corresponds to the device used for user input in the boot services environment. Typically the system keyboard.

ConsoleOut The device handle that corresponds to the device used to display messages to the user from the boot services environment. Typically a display screen.

Desktop Management Interface (DMI)
A platform management information framework, built by the **DMTF** and designed to provide manageability for desktop and server computing platforms by providing an interface that is:
(1) independent of any specific desktop operating system, network operating system, network protocol, management protocol, processor, or hardware platform;
(2) easy for vendors to implement; and
(3) easily mapped to higher-level protocols.

Desktop Management Task Force (DMTF)
The DMTF is a standards organization comprised of companies from all areas of the computer industry. Its purpose is to create the standards and infrastructure for cost-effective management of PC systems.

Device Handle	A handle points to a list of one or more protocols that can respond to requests for services for a given device referred to by the handle.
Device I/O Protocol	A protocol that is used during boot services to access memory and I/O. Also called the **EFI Device I/O Protocol**.
Device Path Instance	When an environment variable represents multiple devices, it is possible for a device path to contain multiple device paths. An example of this would be the **ConsoleOut** environment variable that consists of both a VGA console and a serial output console. This environment variable would describe a console output stream that would send output to both devices and therefore has a Device Path that consists of two complete device paths. Each of these paths is a device path instance.
Device Path Node	A variable-length generic data structure that is used to build a device path. Nodes are distinguished by type, subtype, length, and path-specific data. See **Device Path**.
Device Path Protocol	A protocol that is used during boot services to provide the information needed to construct and manage Device Paths. Also called the EFI **Device Path Protocol**.
Device Path	A variable-length binary data structure that is composed of variable-length generic device path nodes and is used to define the programmatic path to a logical or physical device. There are six major types of device paths: **Hardware Device Path**, **ACPI Device Path**, **Messaging Device Path**, **Media Device Path**, **BIOS Boot Specification Device Path**, and **End Of Hardware Device Path**.
DHCP	See **Dynamic Host Configuration Protocol**.
Disk I/O Protocol	A protocol that is used during boot services to abstract Block I/O devices to allow non-block-sized I/O operations. Also called the EFI Disk I/O Protocol.
DMI	See **Desktop Management Interface**.
DMTF	See **Desktop Management Task Force**.
Dynamic Host Configuration Protocol (DHCP)	A protocol that is used to get information from a configuration server. DHCP is defined by the **Desktop Management Task Force**, not **EFI**.
EBC Image	Executable EBC image following the PE32+ file format.
EBC	See **EFI Byte Code.**
EFI	Extensible Firmware Interface. An interface between the operating system (OS) and the platform firmware.

EFI Application	Modular code that may be loaded in the boot services environment to accomplish platform specific tasks within that environment. Examples of possible applications might include diagnostics or disaster recovery tools shipped with a platform that run outside the OS environment. Applications may be loaded in accordance with policy implemented by the platform firmware to accomplish a specific task. Control is then returned from the application to the platform firmware.
EFI Byte Code (EBC)	The binary encoding of instructions as output by the EBC C compiler and linker. The EBC image is executed by the interpreter.
EFI Driver	A module of code typically inserted into the firmware via protocol interfaces. Drivers may provide device support during the boot process or they may provide platform services. It is important not to confuse drivers in this specification with OS drivers that load to provide device support once the OS takes control of the platform.
EFI File	A container consisting of a number of blocks that holds an image or a data file within a file system that complies with this specification.
EFI Hard Disk	A hard disk that supports the new EFI partitioning scheme (GUID Partitions).
EFI OS Loader	The first piece of operating system code loaded by the firmware to initiate the OS boot process. This code is loaded at a fixed address and then executed. The OS takes control of the system prior to completing the OS boot process by calling the interface that terminates all boot services.
EFI-compliant	Refers to a platform that complies with this specification.
EFI-conformant	See EFI-compliant.
End of Hardware Device Path	A Device Path which, depending on the subtype, is used to indicate the end of the Device Path instance or Device Path structure.
Enhanced Mode (EM)	The 64-bit architecture extension that makes up part of the Intel® Itanium® architecture.
Event Services	The set of functions used to manage events. Includes `CheckEvent()`, `CreateEvent()`, `CloseEvent()`, `SignalEvent()`, and `WaitForEvent()`.
Event	An EFI data structure that describes an "event"—for example, the expiration of a timer.
FAT File System	The file system on which the EFI file system is based. See File Allocation Table and System Partition.

FAT See **File Allocation Table**.

File Allocation Table (FAT)
A table that is used to identify the clusters that make up a disk file. File allocation tables come in three flavors: FAT12, which uses 12 bits for cluster numbers; FAT16, which uses 16 bits; and FAT32, which allots 32 bits but only uses 28 (the other 4 bits are reserved for future use).

File Handle Protocol
A component of the **File System Protocol**. It provides access to a file or directory. Also called the EFI File Handle Protocol.

File System Protocol
A protocol that is used during boot services to obtain file-based access to a device. It has two parts, a **Simple File System Protocol** that provides a minimal interface for file-type access to a device, and a **File Handle Protocol** that provides access to a file or directory.

Firmware Any software that is included in read-only memory (ROM).

Globally Unique Identifier (GUID)
A 128-bit value used to differentiate services and structures in the boot services environment. The format of a **GUID** is defined in Appendix A. See **Protocol**.

GUID Partition Entry
A data structure that characterizes a **GUID Partition**. Among other things, it specifies the starting and ending LBA of the partition.

GUID Partition Table Header
The header in a **GUID Partition Table**. Among other things, it contains the number of partition entries in the table and the first and last blocks that can be used for the entries.

GUID Partition Table
A data structure that describes a **GUID Partition**. It consists of an **GUID Partition Table Header** and, typically, at least one **GUID Partition Entry**. There are two partition tables on an **EFI Hard Disk**: the Primary Partition Table (located in block 1 of the disk) and a Backup Partition Table (located in the last block of the disk). The Backup Table is a copy of the Primary Table.

GUID Partition A contiguous group of sectors on an **EFI Hard Disk**.

Handle See **Device Handle**.

Hardware Device Path
A Device Path that defines how a hardware device is attached to the resource domain of a system (the resource domain is simply the shared memory, memory mapped I/O, and I/O space of the system).

IA-32	See Intel Architecture-32.
Image Handle	A handle for a loaded image; image handles support the loaded image protocol.
Image Handoff State	The information handed off to a loaded image as it begins execution; it consists of the image's handle and a pointer to the image's system table.
Image Header	The initial set of bytes in a loaded image. They define the image's encoding.
Image Services	The set of functions used to manage EFI images. Includes `LoadImage()`, `StartImage()`, `UnloadImage()`, `Exit()`, `ExitBootServices()`, and `EFI_IMAGE_ENTRY_POINT`.
Image	(1) An executable file stored in a file system that complies with this specification. Images may be drivers, applications or OS loaders. Also called an EFI Image. (2) Executable binary file containing EBC and data. Output by the EBC linker.
Intel Architecture Platform Architecture	A collective term for PC-AT-class computers and other systems based on Intel Architecture processors of all families.
Intel Architecture-32 (IA-32)	The 32-bit and 16-bit architecture described in the *Intel Architecture Software Developer's Manual*. IA-32 is the architecture of the Intel P6 family of processors, which includes the Intel® Pentium® Pro, Pentium II, Pentium III, and Pentium 4 processors.
Intel® Itanium® Architecture	The Intel architecture that has 64-bit instruction capabilities, new performance-enhancing features, and support for the IA-32 instruction set. This architecture is described in the *Itanium™ Architecture Software Developer's Manual*.
Interpreter	The software implementation that decodes EBC binary instructions and executes them on a VM. Also called EBC interpreter.
LAN On Motherboard (LOM)	This is a network device that is built onto the motherboard (or baseboard) of the machine.
Legacy Platform	A platform which, in the interests of providing backward-compatibility, retains obsolete technology.
LFN	See Long File Names.

Little Endian	A memory architecture in which the low-order byte of a multibyte datum is at the lowest address, while the high-order byte is at the highest address. See **Big Endian**.
Load File Protocol	A protocol that is used during boot services to find and load other modules of code.
Loaded Image Protocol	A protocol that is used during boot services to obtain information about a loaded image. Also called the EFI Loaded Image Protocol.
Loaded Image	A file containing executable code. When started, a loaded image is given its image handle and can use it to obtain relevant image data.
LOM	See **LAN On Motherboard.**
Long File Names (LFN)	Refers to an extension to the **FAT File System** that allows file names to be longer than the original standard (eight characters plus a three-character extension).
Machine Check Abort (MCA)	The system management and error correction facilities built into the Intel Itanium processors.
Master Boot Record (MBR)	The data structure that resides on the first sector of a hard disk and defines the partitions on the disk.
MBR	See **Master Boot Record**.
MCA	See **Machine Check Abort**.
Media Device Path	A Device Path that is used to describe the portion of a medium that is being abstracted by a boot service. For example, a Media Device Path could define which partition on a hard drive was being used.
Memory Allocation Services	The set of functions used to allocate and free memory, and to retrieve the memory map. Includes **`AllocatePages()`**, **`FreePages()`**, **`AllocatePool()`**, **`FreePool()`**, and **`GetMemoryMap()`**.

Memory Map A collection of structures that defines the layout and allocation of system memory during the boot process. Drivers and applications that run during the boot process prior to OS control may require memory. The boot services implementation is required to ensure that an appropriate representation of available and allocated memory is communicated to the OS as part of the hand-off of control.

Memory Type One of the memory types defined by EFI for use by the firmware and EFI applications. Among others, there are types for boot services code, boot services data, **runtime services** code, and runtime services data. Some of the types are used for one purpose before `ExitBootServices()` is called and another purpose after.

Messaging Device Path
A Device Path that is used to describe the connection of devices outside the **Coherency Domain** of the system. This type of node can describe physical messaging information (e.g., a SCSI ID) or abstract information (e.g., networking protocol IP addresses).

Miscellaneous Services
Various functions that are needed to support the **EFI** environment. Includes `InstallConfigurationTable()`, `ResetSystem()`, `Stall()`, `SetWatchdogTimer()`, `GetNextMonotonicCount()`, and `GetNextHighMonotonicCount()`.

MTFTP See **Multicast Trivial File Transfer Protocol**.

Multicast Trivial File Transfer Protocol (MTFTP)
A **protocol** used to download a **Network Boot Program** to many clients simultaneously from a **TFTP** server.

Name Space In general, a collection of device paths; in an EFI Device Path.

Native Code Low level instructions that are native to the host processor. As such, the processor executes them directly with no overhead of interpretation. Contrast this with **EBC**, which must be interpreted by native code to operate on a **VM**.

NBP See **Network Bootstrap Program** or **Network Boot Program.**

Network Boot Program
A remote boot image downloaded by a **PXE** client using the **Trivial File Transfer Protocol** or the **Multicast Trivial File Transfer Protocol**. See **Network Bootstrap Program.**

Network Bootstrap Program (NBP)

This is the first program that is downloaded into a machine that has selected a PXE capable device for remote boot services.

A typical NBP examines the machine it is running on to try to determine if the machine is capable of running the next layer (OS or application). If the machine is not capable of running the next layer, control is returned to the **EFI** boot manager and the next boot device is selected. If the machine is capable, the next layer is downloaded and control can then be passed to the downloaded program.

Though most NBPs are OS loaders, NBPs can be written to be standalone applications such as diagnostics, backup/restore, remote management agents, browsers, etc.

Network Interface Card (NIC)

Technically, this is a network device that is inserted into a bus on the motherboard or in an expansion board. For the purposes of this document, the term NIC will be used in a generic sense, meaning any device that enables a network connection (including **LOM**s and network devices on external buses (USB, 1394, etc.)).

NIC See **Network Interface Card**.

Page Memory A set of contiguous pages. Page memory is allocated by `AllocatePages()` and returned by `FreePages()`.

Partition Discovery

The process of scanning a block device to determine whether it contains a **Partition**.

Partition See **System Partition**.

PC-AT Refers to a PC platform that uses the AT form factor for their motherboards.

PCI Bus Driver Software that creates a handle for every **PCI controller** on a **PCI Host Bus Controller** and installs both the **PCI I/O Protocol** and the **Device Path Protocol** onto that handle. It may optionally perform **PCI Enumeration** if resources have not already been allocated to all the PCI Controllers on a PCI Host Bus Controller. It also loads and starts any EFI drivers found in any PCI Option ROMs discovered during PCI Enumeration. If a driver is found in a **PCI Option ROM**, the **PCI Bus Driver** will also attach the Bus Specific Driver Override Protocol to the handle for the PCI Controller that is associated with the PCI Option ROM that the driver was loaded from.

PCI Bus A collection of up to 32 physical **PCI Devices** that share the same physical PCI bus. All devices on a PCI Bus share the same **PCI Configuration Space**.

PCI Configuration Space
The configuration channel defined by PCI to configure **PCI Devices** into the resource domain of the system. Each PCI device must produce a standard set of registers in the form of a PCI Configuration Header, and can optionally produce device specific registers. The registers are addressed via Type 0 or Type 1 PCI Configuration Cycles as described by the *PCI Specification*. The PCI Configuration Space can be shared across multiple **PCI Buses**. On most **PC-AT** architecture systems and typical Intel chipsets, the PCI Configuration Space is accessed via I/O ports 0xCF8 and 0xCFC. Many other implementations are possible.

PCI Controller A hardware components that is discovered by a **PCI Bus Driver**, and is managed by a **PCI Device Driver**. **PCI Function** and **PCI Controller** are used equivalently in this document.

PCI Device Driver Software that manages one or more PCI Controllers of a specific type. A driver will use the **PCI I/O Protocol** to produce a device I/O abstraction in the form of another protocol (i.e. Block I/O, Simple Network, Simple Input, Simple Text Output, Serial I/O, Load File).

PCI Device A collection of up to 8 **PCI Functions** that share the same **PCI Configuration Space**. A PCI Device is physically connected to a **PCI bus**.

PCI Enumeration The process of assigning resources to all the PCI Controllers on a given **PCI Host Bus Controller**. This includes PCI Bus Number assignments, PCI Interrupt assignments, PCI I/O resource allocation, the PCI Memory resource allocation, the PCI Prefetchable Memory resource allocation, and miscellaneous PCI DMA settings.

PCI Function A controller that provides some type of I/O services. It consumes some combination of PCI I/O, PCI Memory, and PCI Prefetchable Memory regions, and up to 256 bytes of the **PCI Configuration Space**. The PCI Function is the basic unit of configuration for PCI.

PCI Host Bus Controller
A chipset component that produces PCI I/O, PCI Memory, and PCI Prefetchable Memory regions in a single Coherency Domain. A PCI Host Bus Controller is composed of one or more **PCI Root Bridges**.

PCI I/O Protocol A software interface that provides access to PCI Memory, PCI I/O, and PCI Configuration spaces for a PCI Controller. It also provides an abstraction for PCI Bus Master DMA.

PCI Option ROM A ROM device that is accessed through a PCI Controller, and is described in the PCI Controller's Configuration Header. It may contain one or more **PCI Device Drivers** that are used to manage the PCI Controller.

PCI Root Bridge I/O Protocol
A software abstraction that provides access to the PCI I/O, PCI Memory, and PCI Prefetchable Memory regions in a single Coherency Domain.

PCI Root Bridge A chipset component(s) that produces a physical PCI Local Bus.

PCI Segment A collection of up to 256 **PCI Buses** that share the same **PCI Configuration Space**. PCI Segment is defined in section 6.5.6 of the *ACPI 2.0 Specification* as the _SEG object. The SAL_PCI_CONFIG_READ and SAL_PCI_CONFIG_WRITE procedures defined in chapter 9 of the *SAL Specification* define how to access the PCI Configuration Space in a system that supports multiple PCI Segments. If a system only supports a single PCI Segment the PCI Segment number is defined to be zero. The existence of PCI Segments enables the construction of systems with greater than 256 PCI buses.

Pool Memory A set of contiguous bytes. A pool begins on, but need not end on, an "8-byte" boundary. Pool memory is allocated in pages—that is, firmware allocates enough contiguous pages to contain the number of bytes specified in the allocation request. Hence, a pool can be contained within a single page or extend across multiple pages. Pool memory is allocated by **`AllocatePool()`** and returned by **`FreePool()`**.

Preboot Execution Environment (PXE)
A means by which agents can be loaded remotely onto systems to perform management tasks in the absence of a running OS. To enable the interoperability of clients and downloaded bootstrap programs, the client preboot code must provide a set of services for use by a downloaded bootstrap. It also must ensure certain aspects of the client state at the point in time when the bootstrap begins executing.

The complete PXE specification covers three areas; the client, the network and the server.

Client
- Makes network devices into bootable devices.
- Provides APIs for PXE protocol modules in **EFI** and for universal drivers in the OS.

Network
- Uses existing technology: **DHCP**, **TFTP**, etc.
- Adds "vendor-specific" tags to DHCP to define PXE-specific operation within DHCP.
- Adds multicast TFTP for high bandwidth remote boot applications.
- Defines Bootserver discovery based on DHCP packet format.

Server
Bootserver: Responds to Bootserver discovery requests and serves up remote boot images.
proxyDHCP: Used to ease the transition of PXE clients and servers into existing network infrastructure. proxyDHCP provides the additional

DHCP information that is needed by PXE clients and Bootservers without making changes to existing DHCP servers.

MTFTP: Adds multicast support to a TFTP server.

Plug-In Modules: Example proxyDHCP and Bootservers provided in the PXE SDK (software development kit) have the ability to take plug-in modules (PIMs). These PIMs are used to change/enhance the capabilities of the proxyDHCP and Bootservers.

Protocol Handler Services
The set of functions used to manipulate handles, protocols, and protocol interfaces. Includes `InstallProtocolInterface()`, `UninstallProtocolInterface()`, `ReinstallProtocolInterface()`, `HandleProtocol()`, `RegisterProtocolNotify()`, `LocateHandle()`, and `LocateDevicePath()`.

Protocol Handler A function that responds to a call to a `HandleProtocol` request for a given handle. A protocol handler returns a protocol interface structure.

Protocol Interface Structure
The set of data definitions and functions used to access a particular type of device. For example, BLOCK_IO is a protocol that encompasses interfaces to read and write blocks from mass storage devices. See **Protocol**.

Protocol Revision Number
The revision number associated with a protocol. See **Protocol**.

Protocol The information that defines how to access a certain type of device during boot services. A protocol consists of a **GUID**, a protocol revision number, and a protocol interface structure. The interface structure contains data definitions and a set of functions for accessing the device. A device can have multiple protocols. Each protocol is accessible through the device's handle.

PXE Base Code Protocol
A protocol that is used to control PXE-compatible devices. It is layered on top of a **Simple Network Protocol** to perform packet-level transactions, and may be used by the firmware's boot manager to support booting from remote locations. Also called the EFI PXE Base Code Protocol.

PXE See **Preboot Execution Environment**.

Read-Only Memory (ROM)
When used with reference to the **UNDI** specification, ROM refers to a nonvolatile memory storage device on a **NIC**.

ROM See **Read-Only Memory.**

Runtime Services Driver
A program that is loaded into runtime services memory and stays resident during runtime.

Runtime Services Table
A table that contains the firmware entry points for accessing runtime services functions such as **Time Services** and **Virtual Memory Services**. The table is accessed through a pointer in the **System Table**.

Runtime Services Interfaces that provide access to underlying platform specific hardware that may be useful during OS runtime, such as timers. These services are available during the boot process but also persist after the OS loader terminates boot services.

SAL See **System Abstraction Layer**.

Serial I/O Protocol
A **protocol** that is used during boot services to abstract byte stream devices—that is, to communicate with character-based I/O devices.

Simple File System Protocol
A component of the **File System Protocol**. It provides a minimal interface for file-type access to a device.

Simple Input Protocol
A **protocol** that is used to obtain input from the ConsoleIn device. It is one of two protocols that make up the **Console I/O Protocol**.

Simple Network Protocol
A protocol that is used to provide a packet-level interface to a network adapter. Also called the EFI Simple Network Protocol.

Simple Text Output Protocol
A protocol that is used to control text-based output devices. It is one of two protocols that make up the **Console I/O Protocol**.

SMBIOS See **System Management BIOS**.

StandardError The device handle that corresponds to the device used to display error messages to the user from the boot services environment.

Status Codes Success, error, and warning codes returned by boot services and runtime services functions.

String All strings in this specification are implemented in **Unicode**.

System Abstraction Layer (SAL)
Firmware that abstracts platform implementation differences, and provides the basic platform software interface to all higher level software.

System Management BIOS (SMBIOS)

A table-based interface that is required by the *Wired for Management Baseline Specification*. It is used to relate platform-specific management information to the OS or to an OS-based management agent.

System Partition A section of a block device that is treated as a logical whole. For a hard disk with a legacy partitioning scheme, it is a contiguous grouping of sectors whose starting sector and size are defined by the **Master Boot Record**. For an **EFI Hard Disk**, it is a contiguous grouping of sectors whose starting sector and size are defined by the **GUID Partition Table Header** and the associated **GUID Partition Entries**. For "El Torito" devices, it is a logical device volume. For a diskette (floppy) drive, it is defined to be the entire medium (the term "diskette" includes legacy 3.5" diskette drives as well as newer media such as the Iomega Zip drive). System Partitions can reside on any medium that is supported by EFI boot services. System Partitions support backward compatibility with legacy Intel Architecture systems by reserving the first block (sector) of the partition for compatibility code.

System Table Table that contains the standard input and output handles for an EFI application, as well as pointers to the boot services and runtime services tables. It may also contain pointers to other standard tables such as the **ACPI**, **SMBIOS**, and **SAL** System tables. A loaded image receives a pointer to its system table when it begins execution. Also called the EFI System Table.

Task Priority Level (TPL)

The boot services environment exposes three task priority levels: "normal," "callback," and "notify."

Task Priority Services

The set of functions used to manipulate task priority levels. Includes `RaiseTPL()` and `RestoreTPL()`.

TFTP See **Trivial File Transport Protocol**.

Time Format The format for expressing time in an **EFI-compliant** platform. For more information, see Appendix A.

Time Services The set of functions used to manage time. Includes `GetTime()`, `SetTime()`, `GetWakeupTime()`, and `SetWakeupTime()`.

Timer Services The set of functions used to manipulate timers. Contains a single function, `SetTimer()`.

TPL See **Task Priority Level**.

Trivial File Transport Protocol (TFTP)

A protocol used to download a **Network Boot Program** from a TFTP server.

UNDI	See **Universal Network Device Interface.**
Unicode Collation Protocol	A protocol that is used during boot services to perform case-insensitive comparisons of Unicode strings.
Unicode	An industry standard internationalized character set used for human readable message display.
Universal Network Device Interface (UNDI)	UNDI is an architectural interface to **NIC**s. Traditionally NICs have had custom interfaces and custom drivers (each NIC had a driver for each OS on each platform architecture). Two variations of UNDI are defined in this specification: H/W UNDI and S/W UNDI. H/W UNDI is an architectural hardware interface to a NIC. S/W UNDI is a software implementation of the H/W UNDI.
Universal Serial Bus (USB)	A bi-directional, isochronous, dynamically attachable serial interface for adding peripheral devices such as serial ports, parallel ports, and input devices on a single bus.
USB Bus Driver	Software that enumerates and creates a handle for each newly attached USB Controller and installs both the **USB I/O Protocol** and the Device Path Protocol onto that handle, starts that device driver if applicable. For each newly detached USB Controller, the device driver is stopped, the USB I/O Protocol and the Device Path Protocol are uninstalled from the device handle, and the device handle is destroyed.
USB Bus	A collection of up to 127 physical **USB Devices** that share the same physical USB bus. All devices on a USB Bus share the bandwidth of the USB Bus.
USB Controller	A hardware component that is discovered by a **USB Bus Driver**, and is managed by a **USB Device Driver**. **USB Interface** and **USB Controller** are used equivalently in this document.
USB Device Driver	Software that manages one or more **USB Controller** of a specific type. A driver will use the **USB I/O Protocol** to produce a device I/O abstraction in the form of another protocol (i.e. Block I/O, Simple Network, Simple Input, Simple Text Output, Serial I/O, Load File).
USB Device	A USB peripheral that is physically attached to the **USB Bus**.
USB Enumeration	A periodical process to search the **USB Bus** to detect if there have been any **USB Controller** attached or detached. If an attach event is detected, then the USB Controllers device address is assigned, and a child handle is created. If a detach event is detected, then the child handle is destroyed.

USB Host Controller
Moves data between system memory and devices on the **USB Bus** by processing data structures and generating the USB transactions. For USB 1.1, there are currently two types of USB Host Controllers: UHCI and OHCI.

USB Hub A special **USB Device** through which more USB devices can be attached to the **USB Bus**.

USB I/O Protocol A software interface that provides services to manage a **USB Controller**, and services to move data between a USB Controller and system memory.

USB Interface The USB Interface is the basic unit of a physical **USB Device**.

USB See **Universal Serial Bus**.

Variable Services The set of functions used to manage variables. Includes `GetVariable()`, `SetVariable()`, and `GetNextVariableName()`.

Virtual Memory Services
The set of functions used to manage virtual memory. Includes `SetVirtualAddressMap()` and `ConvertPointer()`.

VM The Virtual Machine, a pseudo processor implementation consisting of registers which are manipulated by the interpreter when executing **EBC** instructions.

Watchdog Timer An alarm timer that may be set to go off. This can be used to regain control in cases where a code path in the boot services environment fails to or is unable to return control by the expected path.

WfM See **Wired for Management**.

Wired for Management (WfM)
Refers to the *Wired for Management Baseline Specification*. The Specification defines a baseline for system manageability issues; its intent is to help lower the cost of computer ownership.

Index