intel®

# Extensible Firmware Interface Specification

Version 1.02

December 12, 2000

Intel order number 731843-001

# Revision History

| Revision | Revision History | Date |
|---|---|---|
| 1.01 | Original Issue. | 12/01/00 |
| 1.02 | Update for legal and trademarking requirements. | 12/12/00 |

# Table of Contents

# 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 and its loader. 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 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 Overview

This specification is organized as follows:

**Table 1-1. Organization of EFI Specification**

| Chapter/Appendix | Description |
|---|---|
| 1. Introduction | Provides an overview of the EFI Specification. |
| 2. Overview | Describes the major components of EFI, including the boot manager, firmware core, calling conventions, protocols, and requirements. |
| 3. Services | Contains definitions for the fundamental services that are present in an EFI-compliant system. |
| 4. EFI Image | Defines EFI images, a class of files that contain executable code. |
| 5. Device Path Protocol | Defines the device path protocol and provides the information needed to construct and manage device paths in the EFI environment. |
| 6. 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. |
| 7. Console I/O Protocol | Defines the Console I/O protocol, which handles input and output of text-based information intended for the system user while executing in the EFI boot services environment. |
| 8. Block I/O Protocol | Defines the Block I/O protocol, which is used to abstract mass storage devices to allow code running in the EFI boot services environment to access the devices without specific knowledge of the type of device or controller that manages the device. |
| 9. Disk I/O Protocol | Defines the Disk I/O protocol, which is used to abstract Block I/O devices to allow non-block sized I/O operations. |
| 10. File System Protocol | Defines the File System protocol, which allows code running in the EFI boot services environment to obtain file based access to a device. |
| 11. Load File Protocol | Defines the Load File protocol, which allows code running in the EFI boot services environment to find and load other modules of code. |
| 12. Serial I/O Protocol | Defines the Serial I/O protocol, which is used to abstract byte stream devices. |
| 13. Unicode Collation Protocol | Defines the Unicode Collation protocol, which is used to allow code running in the EFI boot services environment to perform lexical comparison functions on Unicode strings for given languages. |
| 14. PXE Base Code Protocol | Defines the PXE Base Code protocol, which is used perform network boot operations. |

continued

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

| Chapter/Appendix | Description |
|---|---|
| 15. Simple Network Protocol | Defines the Simple Network Protocol, which provides a packet level interface to a network device. Also defines the Network Interface Identifier Protocol, which is an optional protocol used to describe details about the software layer used to produce the Simple Network Protocol. |
| 16. File System Format | Defines the EFI file system. |
| 17. Boot Manager | Describes the boot manager, which is used to load EFI drivers and EFI applications. |
| 18. PCI Expansion ROM | Describes how to provide an EFI driver image within a PCI expansion ROM. |
| 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. Alphabetic Function List | Lists all EFI interface functions alphabetically. |
| F. Glossary | Briefly describes terms defined or referenced by this specification. |
| G. 32/64-Bit UNDI Specification | This appendix defines the 32/64-bit H/W and S/W Universal Network Driver Interfaces (UNDIs). |
| H. Index | Provides an index to the key terms and concepts in the specification. |

## 1.2 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.
- *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 non-volatile 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.

## 1.3 Target Audience

This document is intended for the following readers:

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

## 1.4 Related Information

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

- *ACPI Implementers' Guide*, Intel Corporation, Microsoft Corporation, Toshiba Corporation, version 0.5, 1998, http://www.teleport.com/~acpi
- *Advanced Configuration and Power Interface Specification*, http://www.teleport.com/~acpi
- *BIOS Boot Specification Version 1.01*, Compaq Computer Corporation, Phoenix Technologies Ltd., Intel Corporation, 1996, http://www.phoenix.com/products/specs.html
- *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/respec/pnpspecs.htm
- *"El Torito" Bootable CD-ROM Format Specification,* Version 1.0, Phoenix Technologies, Ltd., IBM Corporation, 1994, http://www.phoenix.com/products/specs.html
- *File Verification Using CRC*, Mark R. Nelson, Dr. Dobbs, May 1994
- *Hardware Design Guide Version 2.0 for Microsoft Windows NT† Server*, Intel Corporation, Microsoft Corporation, 1998, http://developer.intel.com/design/servers/desguide/index.htm
- *Intel Architecture Software Developer's Manual*, http://developer.intel.com/design/mmx/manuals
- *IA-64 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/ia-64
- *IA-64 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/ia-64
- *IA-64 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/ia-64
- *IA-64 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/ia-64
- *IA-64 Software Conventions and Runtime Architecture Guide,* Order number 245358, Intel Corporation, January, 2000. Also available at http://developer.intel.com/design/ia-64
- *IA-64 System Abstraction Layer Specification,* Available at http://developer.intel.com/design/ia-64
- *IEEE 1394 Specification*, http://www.1394ta.org/
- 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
- *OSTA Universal Disk Format Specification*, Revision 2.00, Optical Storage Technology Association, 1998, http://www2.osta.org/osta/html/ostatech.html#udf
- *PCI BIOS Specification,* Revision 2.1, PCI Special Interest Group, Hillsboro, OR, http://www.pcisig.com/tech/index.html
- *PCI Local Bus Specification* Revision 2.2, PCI Special Interest Group, Hillsboro, OR, http://www.pcisig.com/tech/index.html
- *Portable Executable and Common Object File Format Specification*. See http://msdn.microsoft.com/library/specs/msdn_pecoff.htm
- *Preboot Execution Environment (PXE) Specification*, Version 2.1. Intel Corporation, 1999. Available at ftp://download.intel.com/ial/wfm/pxespec.pdf.
- *Plug and Play BIOS Specification*, Version 1.0A, Compaq Computer Corporation, Phoenix Technologies, Ltd., Intel Corporation, 1994, ftp://download.intel.com/ial/wfm/bio10a.pdf or http://www.microsoft.com/hwdev/respec/pnpspecs.htm
- *POST Memory Manager Specification,* Version 1.01, Phoenix Technologies Ltd., Intel Corporation, 1997, http://www.phoenix.com/products/specs.html
- [RFC 791] *Internet Protocol DARPA Internet Program Protocol (IPv4) Specification*, September 1981, http://www.faqs.org/rfcs/rfc791.html
- [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
- *SYSID BIOS Support Interface Requirements,* Version 1.2, Intel Corporation, 1997, http://developer.intel.com/ial/WfM/design/mapxe/pxespec.htm
- *SYSID Programming Interface* Version 1.2, http://developer.intel.com/ial/WfM/design/mapxe/pxespec.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://developer.intel.com/ial/WfM/design/BIBLIOG.HTM or http://www.phoenix.com/products/specs.html
- *The Unicode Standard,* Version 2.1, Unicode Consortium, http://www.unicode.org/
- *ISO 639-2:1998.* Codes for the Representation of Names of Languages – Part2: Alpha-3 code, http://www.iso.ch/
- *Universal Serial Bus PC Legacy Compatibility Specification*, Version 0.9, http://www.usb.org/developers/index.html
- *Wired for Management Baseline,* Version 2.0 Release Candidate. Intel Corporation, 1998, http://developer.intel.com/ial/WfM

## 1.5 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.

### 1.5.1 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.teleport.com/~acpi).

### 1.5.2 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://developer.intel.com/ial/WfM. To obtain the *System Management BIOS Reference Specification*, visit http://developer.intel.com/ial/WfM/design/BIBLIOG.HTM.

### 1.5.3 Additional Considerations for Itanium™-based Platforms

Any information or service that is available via Itanium-based firmware architecture specifications supercedes any requirement in the common IA-32 and Itanium-based specifications listed above. The Itanium-based firmware architecture specifications (currently the *IA-64 System Abstraction Layer Specification* and portions of the *IA-64 Architecture Software Developer's Manual, Volumes 1-4*) define the baseline functionality required for all Itanium-based platforms. The major addition that EFI makes to these Itanium-based firmware architecture specifications is that it defines a boot infrastructure and a set of services that constitute a common platform definition for high volume Itanium-based systems to implement based on the more generalized Itanium-based firmware architecture specifications.

The following specifications are the required Intel Itanium architecture specifications for all Itanium-based platforms:

- *IA-64 System Abstraction Layer Specification*.
- *IA-64 Architecture Software Developer's Manual, Volumes 1-4*.

Both documents are available at http://developer.intel.com/design/ia-64.

## 1.6 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 EI specification. (See Section 1.5 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 non-volatile platform memory.
- *Boot services*. Boot Services provides 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.

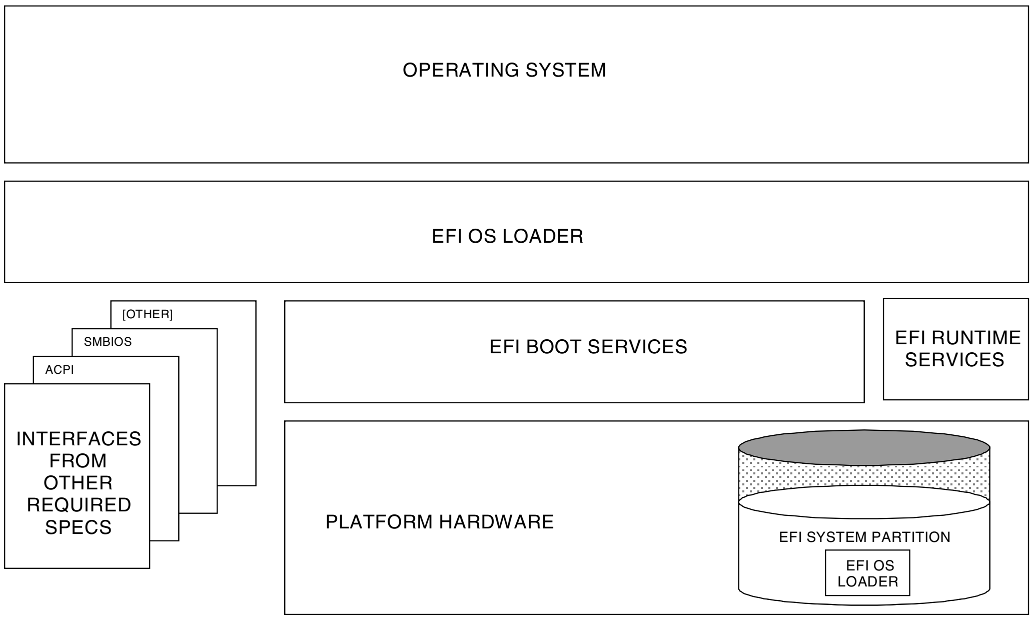

**Figure 1-1. EFI Conceptual Overview**

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.

## 1.7 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

The Intel architecture processors of the IA-32 family are “little endian” machines. This means that the low-order byte of a multi-byte data item in memory is at the lowest address, while the high-order byte is at the highest address. 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 Typographic Conventions

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

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

## 1.9 Guidelines for Use of the Term “Extensible Firmware Interface”

The following recommendations are offered for developers creating products or documentation that have the need to make reference to this specification. The intent of these recommendations is to ensure consistent usage of the name of the specification in the industry and to avoid ambiguous or out of context use that might otherwise cause confusion.

- In any given piece of collateral or other materials where it is necessary to abbreviate “Extensible Firmware Interface,:” the first use and all prominent uses of “Extensible Firmware Interface” should be fully spelled out and immediately followed by “EFI” in parentheses. This will establish that EFI is being used only as an abbreviation. For example, “The Extensible Firmware Interface (hereafter “EFI”) is an architecture specification.”
- After the first use where “Extensible Firmware Interface” is fully spelled out, “EFI” may be used standalone. As indicated above, the abbreviation should only be used where necessary, e.g. where space constrained or to avoid excessive repetition in text and the abbreviation should not be used in any prominent places, such as in titles or paragraph headings.

# 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.

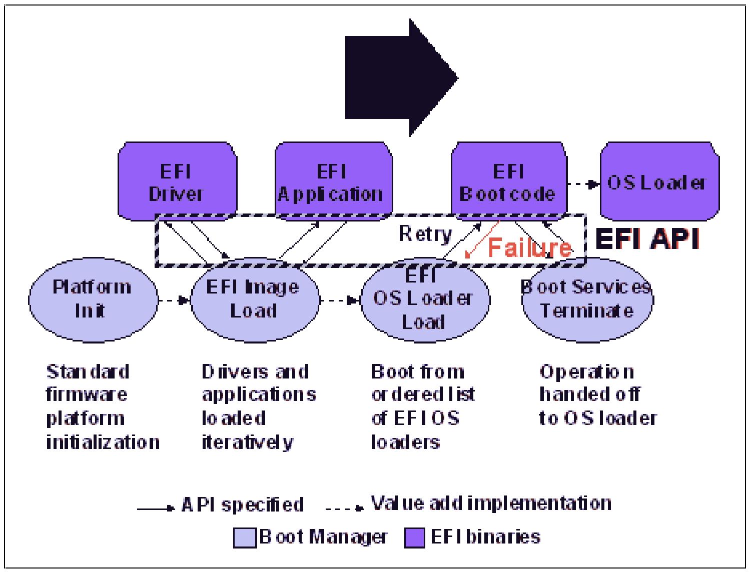

**Figure 2-1. Booting Sequence**

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

## 2.2 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 sub-categories:
  - 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 **`SetVirtualAddressMap()`** service, the OS will only be able to call EFI runtime services in a virtual addressing mode. All runtime interfaces are non-blocking 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 those resources are never used by the OS.

The following table 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. |
| GetNextVariableByName | 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.

IA-32 and Itanium-based calling conventions 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-based operations) |
| UINTN | Unsigned value of native width. (4 bytes on IA-32, 8 bytes on Itanium-based operations) |
| INT8 | 1 byte signed value. |
| UINT8 | 1 byte unsigned value. |
| INT16 | 2 byte signed value. |
| UINT16 | 2 byte unsigned value. |
| INT32 | 4 byte signed value. |
| UINT32 | 4 byte unsigned value. |
| INT64 | 8 byte signed value. |

continued

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

| Mnemonic | Description |
|---|---|
| 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 an enumeration. 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. |
| UNALIGNED | Datum is byte packed and is not naturally aligned. |
| 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 non-volatile 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.

## 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.

Refer to the *IA-64 System Abstraction Layer Specification* 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* for more information.

## 2.4 Protocols

The protocols that a device handle supports are discovered through the **`HandleProtocol()`** service. Each protocol has a specification that includes:

- 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()`**. 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 2-2 shows the construction of a protocol. The EFI driver contains functions specific to one or more protocol implementations, and registers them with **`InstallProtocolInterface()`** service. 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.

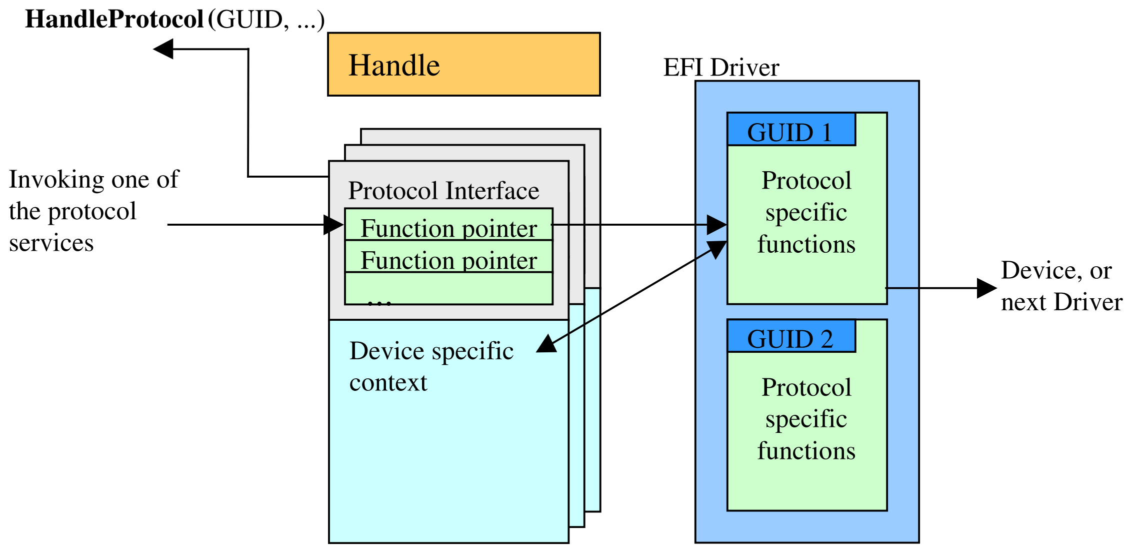

**Figure 2-2. Construction of a Protocol**

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 Name | Description |
|---|---|
| BLOCK_IO | Protocol interfaces for devices that support block I/O style accesses. |
| DEVICE_IO | Protocol interfaces for performing device I/O. |
| DEVICE_PATH | Provides the location of the device. |
| DISK_IO | A protocol interface that layers onto any BLOCK_IO interface. |
| SIMPLE_FILE_SYSTEM | Protocol interfaces for opening disk volume containing an EFI file system. |
| EFI_FILE_HANDLE | Provides access to supported file systems. |
| LOAD_FILE | Protocol interface for reading a file from an arbitrary device. |
| LOADED_IMAGE | Provides information on the image. |
| PXE_BC | Protocol interfaces for devices that support network booting. |
| SERIAL_IO | Protocol interfaces for devices that support serial character transfer. |
| 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. |
| SIMPLE_NETWORK | Provides interface for devices that support packet based transfers. |
| UNICODE_COLLATION | Protocol interfaces for Unicode string comparison operations. |

## 2.5 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.

### 2.5.1 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. A system must provide the LOAD_FILE protocol or the SIMPLE_FILE_SYSTEM protocol or both. It is possible for a system to boot an OS using just the LOAD_FILE protocol. In this case, the SIMPLE_FILE_SYSTEM, EFI_FILE_HANDLE, DISK_IO, and BLOCK_IO protocols would not be required.

**Table 2-5. Required EFI Implementation Elements**

| Element | Description |
|---|---|
| Boot Services | All functions defined as boot services. |
| Runtime Services | All functions defined as runtime services. |
| Partitioning[1] | Functionality to provide BLOCK_IO interfaces for logical block devices as defined by partition table, or El Torito “no emulation” device. |
| BLOCK_IO protocol[1] | Protocol interfaces for devices that support block I/O style accesses. |
| DEVICE_IO protocol | Protocol interfaces for performing device I/O. |
| DEVICE_PATH protocol | Provides the location of the device. |
| DISK_IO protocol[1] | Protocol interfaces for providing disk IO from a BLOCK_IO interface. |
| LOAD_FILE protocol | Protocol interface for reading a file from an arbitrary device. |
| LOADED_IMAGE protocol | Provides information on the image. |
| SIMPLE_FILE_SYSTEM protocol[1] | Protocol interfaces for opening disk volumes through a DISK_IO interface. |
| EFI_FILE_HANDLE protocol[1] | Protocol interfaces for accessing the device with file I/O style accesses through a DISK_IO interface. |

continued

**Table 2-5. Required EFI Implementation Elements** (continued)

| Element | Description |
|---|---|
| SIMPLE_INPUT protocol | Protocol interfaces for devices that support simple console style text input. |
| SIMPLE_TEXT_OUTPUT protocol | Protocol interfaces for devices that support console style text displaying. |
| UNICODE_COLLATION protocol[1] | Protocol interfaces for Unicode string comparison operations. |

[1] These protocols are not required if the implementation can operate using on the LOAD_FILE protocol.

## 2.5.2 Optional Elements

Table 2-6 lists the optional elements. Any system that is designed to conform to the EFI specification *may choose* whether or not to provide a complete implementation of all these elements. *However*, any system choosing to provide an implementation of one of these optional elements must do so to the same extent as for required elements. In other words, an implementation of any single optional element of this specification must include all the functions defined as part of the option and must deliver the full semantics defined for the services for all combinations of calls and parameters.

**Table 2-6. Optional EFI Implementation Elements**

| Element | Description |
|---|---|
| SERIAL_IO protocol | Protocol interfaces for byte stream devices. |
| SIMPLE_NETWORK protocol | Protocol interfaces for devices that support packet based transfers. |
| PXE_BC protocol | Protocol interfaces for devices that support PXE I/O network access. |
| Partitioning[1] | Functionality to provide BLOCK_IO interfaces for logical block devices as defined by partition table, or El Torito “no emulation” device. |
| BLOCK_IO protocol[1] | Protocol interfaces for devices that support block I/O style accesses. |
| DISK_IO protocol[1] | Protocol interfaces for providing disk IO from a BLOCK_IO interface. |
| SIMPLE_FILE_SYSTEM protocol[1] | Protocol interfaces for opening disk volumes through a DISK_IO interface. |
| EFI_FILE_HANDLE protocol[1] | Protocol interfaces for accessing the device with file I/O style accesses through a DISK_IO interface. |
| UNICODE_COLLATION protocol[1] | Protocol interfaces for Unicode string comparison operations. |

[1] These protocols are not optional if the implementation requires them to support SIMPLE_FILE_SYSTEM protocol.

### 2.5.3 Appendixes

The content of Appendixes B, C, D, E of this specification is largely intended as informational. In other words, semantic information contained in these sections need not be considered part of the formal definition of either required or optional elements of the specification.

The content of Appendix A is a set of definitions that are used extensively by other interfaces in the specification. As such, implementations are required to use the time and GUID formats defined therein.

# 3 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 pre-boot environment and EFI OS loaders.

Two types of services are described here:

- **Boot Services**. Functions that are available *before* a successful call to `ExitBootServices()`.
- **Runtime Services**. Functions that are available *before and after* any call to `ExitBootServices()`.

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 3.1)
- Memory Allocation Services (Section 3.2)
- Protocol Handler Services (Section 3.3)
- Image Services (Section 3.4)
- Miscellaneous Services (Section 3.8)

Runtime Services fall into these categories:

- Variable Services (Section 3.5)
- Time Services (Section 3.6)
- Virtual Memory Services (Section 3.7)
- Miscellaneous Services (Section 3.8)

## 3.1 Event, Timer, and Task Priority Services

The functions that make up the Event, Timer, and Task Priority Services are used during pre-boot to create, close, signal, and wait for events; to set timers; and to raise and restore task priority levels. See Table 3-1.

**Table 3-1. 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 3-2 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 3-2. 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 execting 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. |

continued

**Table 3-2. TPL Usage** (continued)

| Task Priority Level | Usage |
|---|---|
| (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 3-3 summarizes the restrictions.

**Table 3-3. TPL Restrictions**

| Name | Restriction | Task Priority Level |
|---|---|---|
| Memory Allocation Services | <= | **TPL_NOTIFY** |
| Variable Services | <= | **TPL_CALLBACK** |
| ExitBootServices() | = | **TPL_APPLICATION** |
| LoadImage() | <= | **TPL_CALLBACK** |
| WaitForEvent() | = | **TPL_APPLICATION** |
| SignalEvent() | <= | **TPL_HIGH_LEVEL** |
| Event Notification Levels | ><br><= | **TPL_APPLICATION**<br>**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** |

## 3.1.1 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 Section 3.1.7. |
| *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()** (Section 3.7.1). |
| **EVT_RUNTIME_CONTEXT** | The event's *NotifyContext* pointer points to a runtime memory address. See the discussion of **EVT_RUNTIME**. |
| **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 can not 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 can not 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 3.1 (**TPL_APPLICATION**, **TPL_CALLBACK**, and **TPL_NOTIFY**). The functions in these queues are invoked in FIFO order, starting with the highest priority level queueand 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 can't 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. For more information, see Section 3.7.1.

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: Since 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. |

### 3.1.2 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 Section 3.1.1. |

#### 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. |

### 3.1.3 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 Section 3.1.1. |

#### 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 notificiation task priority level. **SignalEvent()** may be invoked from any task priority level.

#### Status Codes Returned

| | |
|---|---|
| EFI_SUCCESS | The event was signaled. |

### 3.1.4 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 Section 3.1.1. |
| *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**. |

### 3.1.5 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 Section 3.1.1. |

#### 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. If *Event* is of type **EFI_NOTIFY_WAIT**, 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. |

## 3.1.6 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 Section 3.1.1. |
| *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. |

## 3.1.7 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 codeing 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()**.

### 3.1.8 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 Section 3.1.7. |

#### 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 codeing 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.

## 3.2 Memory Allocation Services

The functions that make up Memory Allocation Services are used during pre-boot to allocate and free memory, and to obtain the system's memory map. See Table 3-4.

**Table 3-4. 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 pre-boot, has three major consequences:

1. During pre-boot, 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 pre-boot, 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 Section 3.2.1). Some of the types have a different usage *before* **`ExitBootServices()`** is called than they do *afterwards*. Table 3-5 lists each type and its usage before the call; Table 3-6 lists each type and its usage after the call.

**Table 3-5. 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. |
| EfiPalCode | Address space reserved by the firmware for code that is part of the processor. |
| EfiFirmwareReserved | Address space reserved by the firmware. |

**Table 3-6. 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. |
| EfiFirmwareReserved | In general, this memory is not to be used by the loader or OS; however, specific functions may point to ranges within this memory to be used. |

## 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`**.

## 3.2.1 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 only types allowed are **EfiLoaderCode, EfiLoaderData, EfiRuntimeServicesCode, EfiRuntimeServicesData, EfiBootServicesCode, EfiBootServicesData, EfiACPIReclaimMemory,** and **EfiACPIMemoryNVS**. Normal allocations (that is, allocations by any EFI application) are of type **EfiLoaderData**. See "Related Definitions", Table 3-5, and Table 3-6. |
| *Pages* | The number of contiguous 4KB 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 3-5 and Table 3-6.
 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 | One of the parameters has an invalid value. |

## 3.2.2 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 Section 3.2.1. |
| *Pages* | The number of contiguous 4KB 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. |

### 3.2.3 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 (**EFI_MEMORY_TYPE**, see Section 3.2.1). |
| **PhysicalStart** | Physical address of the first byte in the memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in Section 3.2.1. |
| **VirtualStart** | Virtual address of the first byte in the memory region. Type **EFI_VIRTUAL_ADDRESS** is defined in "Related Definitions". |
| **NumberOfPages** | Number of pages in the memory region. |
| **Attribute** | Attributes of the 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: Memory region is not cacheable. |
| **EFI_MEMORY_WC** | Memory cacheability attribute: Memory region supports write combining. |
| **EFI_MEMORY_WT** | Memory cacheability attribute: Memory region is cacheable with "write through" policy. Writes that hit in the cache will also be written to main memory. |

| | |
|---|---|
| **`EFI_MEMORY_WB`** | Memory cacheability attribute: Memory region is cacheable with "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: Memory region is uncacheable,exported, and supports the "fetch and add" semaphore mechanism. |
| **`EFI_MEMORY_WP`** | Physical memory protection attribute: Memory region is write-protected by system hardware. |
| **`EFI_MEMORY_RP`** | Physical memory protection attribute: Memory region is read-protected by system hardware. |
| **`EFI_MEMORY_XP`** | Physical memory protection attribute: Memory region is protected against executing code by system hardware. |
| **`EFI_MEMORY_RUNTIME`** | Runtime memory attribute: The memory region needs to be given a virtual mapping by the operating system when **`SetVirtualAddressMap()`** is called. |

```
//*******************************************************
//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 which the firmware is using for its own purposes.

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. |

## 3.2.4 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. The only supported types are **EfiLoaderData, EfiBootServicesData, EfiRuntimeServicesData, EfiACPIReclaimMemory,** and **EfiACPIMemoryNVS**. Type **EFI_MEMORY_TYPE** is defined in Section 3.2.1. |
| *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. |

### 3.2.5 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. |

## 3.3 Protocol Handler Services

In the abstract, a protocol consists of a 128-bit guaranteed 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 3-7.

**Table 3-7. 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. |

As depicted in Figure 3-1, 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 3-1 shows that protocol handlers are listed for each device handle and that each protocol handler is logically an EFI driver.

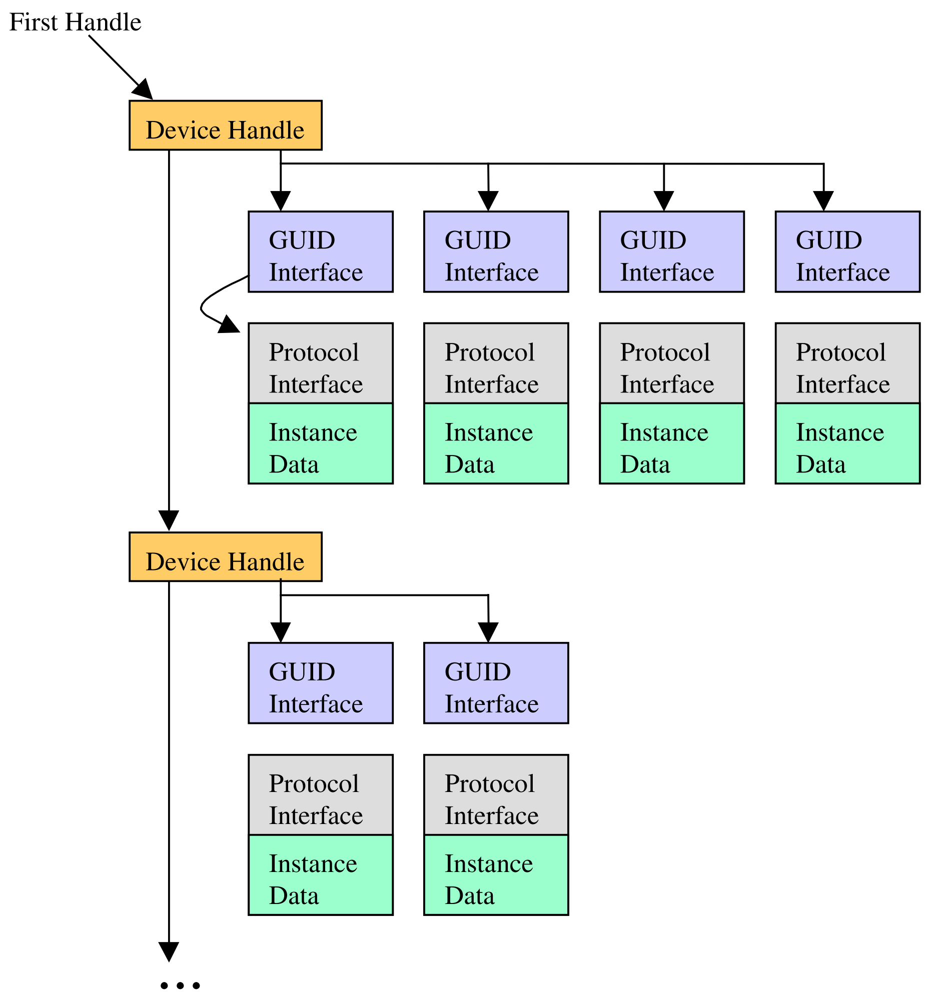

**Figure 3-1. Device Handle to Protocol Handler Mapping**

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.

## 3.3.1 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 or p-code 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_PCODE_INTERFACE
} EFI INTERFACE_TYPE;
```

## Description

The **`InstallProtocolInterface()`** function installs a protocol interface (a GUID/Protocol Interface structure pair) on a device handle.

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 Section 3.3.4.

## 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. |

### 3.3.2 UninstallProtocolInterface()

#### Summary

Removes a protocol interface from a device handle.

#### Prototype

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

#### Parameters

| | |
|---|---|
| *Handle* | The handle on which the interface was installed. Type **EFI_HANDLE** is defined in Section 3.3.1. If *Handle* is not a valid handle, then **EFI_INVALID_PARAMETER** is returned. |
| *Protocol* | The numeric ID of the interface. Type **EFI_GUID** is defined in Section 3.3.1. It is the callers responsibility to pass in a valid GUID. See "Wired For Management Baseline" for a description of valid GUID values. |
| *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.

#### Status Codes Returned

| | |
|---|---|
| EFI_SUCCESS | The interface was removed. |
| EFI_NOT_FOUND | The interface was not found. |
| EFI_INVALID_PARAMETER | One of the parameters has an invalid value. |

## 3.3.3 ReinstallProtocolInterface()

### Summary

Reinstalls a protocol interface on a device handle.

### Prototype

```
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. Type **EFI_HANDLE** is defined in Section 3.3.1. If *Handle* is not a valid handle, then EFI_INVALID_PARAMETER is returned. |
| *Protocol* | The numeric ID of the interface. Type **EFI_GUID** is defined in Section 3.3.1. It is the callers responsibility to pass in a valid GUID. See "Wired For Management Baseline" for a description of valid GUID values. |
| *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. For more information, see Section 3.3.4.

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

### Status Codes Returned

| | |
|---|---|
| EFI_SUCCESS | The protocol interface was installed. |
| EFI_NOT_FOUND | The *OldInterface* on the handle was not found. |
| EFI_INVALID_PARAMETER | One of the parameters has an invalid value. |

## 3.3.4 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 Section 3.3.1. |
| *Event* | Event that is to be signaled whenever a protocol interface is registered for *Protocol*. Type **EFI_EVENT** is defined in Section 3.1.1. |
| *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. |

## 3.3.5 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 Section 3.3.1. |
| *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 **RegisterNotifyProtocol()**. |
| *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 Section 3.3.1. |

## 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. |

### 3.3.6 HandleProtocol()

#### Summary

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

#### Prototype

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

#### Parameters

| | |
|---|---|
| *Handle* | The handle being queried. Type **EFI_HANDLE** is defined in Section 3.3.1. If *Handle* is not a valid **EFI_HANDLE**, then **EFI_INVALID_PARAMETER** is returned. |
| *Protocol* | The published unique identifier of the protocol. Type **EFI_GUID** is defined in Section 3.3.1. 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*. |

#### 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.

#### 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. |

## 3.3.7 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 Section 3.3.1. |
| *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 Section 3.3.1. |

### 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 IO 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. |

## 3.4 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 3-8 summarizes the differences between images.

**Table 3-8. 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().**<br>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 behaviour | 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.<br>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.<br>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 3-9 lists the functions that make up Image Services.

**Table 3-9. 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. |

## 3.4.1 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 Section 3.3.1. This field is used to initialize the *ParentHandle* field of the **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 Section 3.3.7. |
| *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 inSection 3.3.1. |

## 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 **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 **Unload()**.

## 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 can not be parsed to locate the proper protocol for loading the 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. |

### 3.4.2 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 Section 3.3.1. |
| *ExitDataSize* | Pointer to the size, in bytes, of *ExitData*. |
| *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()** (see Section 3.4.5) 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.

#### Status Codes Returned

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

### 3.4.3 UnloadImage()

#### Summary

Unloads an image.

#### Prototype

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

#### Parameters

| | |
|---|---|
| *ImageHandle* | Handle that identifies the image to be unloaded. Type **EFI_HANDLE** is defined in Section 3.3.1. |

#### 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**.

#### 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 image's unload function. |

## 3.4.4 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 Section 3.3.1. |
| *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 **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 **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()** for entry point exit conditions (Section 3.4.5).

## 3.4.5 Exit()

### Summary

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

### Prototype

```
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. Type **EFI_HANDLE** is defined in Section 3.3.1. |
| *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()**. (See Section 3.2.4.) |

### 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 callouts 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.

## Status Codes Returned

| | |
|---|---|
| (Does not return.) | Image exit. Control is returned from 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. |

### 3.4.6 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 Section 3.3.1. |
| *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 serviceswatchdog timer is disabled.

#### Status Codes Returned

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

## 3.5 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 non-volatile 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 3-10 lists the variable services functions described in this section:

**Table 3-10. 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. |

### 3.5.1 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 Section 3.3.1. |
| *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 *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. |
| EFI_DEVICE_ERROR | The variable could not be retrieved due to a hardware error. |

### 3.5.2 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 Section 3.3.1. |

#### 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 | One of the parameters has an invalid value. |
| EFI_DEVICE_ERROR | The variable name could not be retrieved due to a hardware error. |

### 3.5.3 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*. |
| *VendorGuid* | A unique identifier for the vendor. Type **EFI_GUID** is defined in Section 3.3.1. |
| *Attributes* | Attributes bitmask to set for the variable. See Section 3.5.1. |
| *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 non-volatile 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 non-volatile variable when absolutely necessary. In addition, if software uses a non-volatile 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 pre-existing 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 re-create it with different attributes. There is one exception to this rule. If a pre-existing 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 non-volatile are effective read-only data variables once **ExitBootServices()** is performed.

The only rules the firmware must implement when saving a non-volatile variable is that it has actually been saved to non-volatile 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 *VariableSize* exceeds the maximum allowed. |
| 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. |

## 3.6 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 pre-boot environment. Table 3-11 lists the time services functions described in this section:

**Table 3-11. 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. |

### 3.6.1 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.<br><br>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.<br><br>If **EFI_TIME_IN_DAYLIGHT** is set, the time has been adjusted for daylight savings time.<br><br>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 | One of the parameters has an invalid value. |
| EFI_DEVICE_ERROR | The time could not be retrieved due to a hardware error. |

## 3.6.2 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 Section 3.6.1. Full error checking is performed on the different fields of the **EFI_TIME** structure (refer to the **EFI_TIME** definition on page 85 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. |

## 3.6.3 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 Section 3.6.1. |

### 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 | A time field is out of range. |
| EFI_DEVICE_ERROR | The wakeup time could not be retrieved due to a hardware error. |

### 3.6.4 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 Section 3.6.1. 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. |

## 3.7 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 3-12 lists the virtual memory service functions described in this section.

**Table 3-12. 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. |

### 3.7.1 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 Section 3.2.3. |

#### 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.

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 re-applies image "fixup" information to virtually relocate all runtime images to their new addresses.

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. |

## 3.7.2 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. |

## 3.8 Miscellaneous Services

This section contains the remaining function definitions for services not defined elsewhere but which are required to complete the definition of the EFI environment. Table 3-13 lists the Miscellaneous Services Functions.

**Table 3-13. Miscellaneous Services Functions**

| Name | Type | Description |
|---|---|---|
| ResetSystem | Runtime | Resets the entire platform. |
| SetWatchDogTimer | Boot | Resets and sets a watchdog timer used during boot services time. |
| Stall | Boot | Stalls the processor. |
| GetNextMonotonicCount | Boot | Returns a monotonically increasing count for the platform. |
| GetNextHighMonotonicCount | Runtime | Returns the next high 32 bits of the platform's monotonic counter. |
| InstallConfigurationTable | Boot | Adds, updates, or removes a configuration table from the EFI System Table. |

### 3.8.1 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". |
| *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
} 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.

The platform may optionally log the parameters from any non-normal reset that occurs.

The **`SystemReset()`** function does not return.

## 3.8.2 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, a system reset is generated and the event is logged by the firmware. 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. |

### 3.8.3 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*. |

### 3.8.4 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 non-volatile and is increased by 1 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. |

### 3.8.5 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 non-volatile 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 non-volatile 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 | One of the parameters has an invalid value. |

## 3.8.6 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 non-existent entry. |
| EFI_OUT_OF_RESOURCES | There is not enough memory available to complete the operation. |

# 4
# EFI Image

This chapter defines EFI images, a class of files that contain executable code. We begin by describing the **EFI_LOADED_IMAGE** protocol, and then discuss EFI image headers, applications, OS loaders, and drivers.

## 4.1 LOADED_IMAGE Protocol

This section provides a detailed description of the **EFI_LOADED_IMAGE** protocol.

### Summary

Can be used on any image handle to obtain information about the loaded image.

### GUID

```
#define LOADED_IMAGE_PROTOCOL          \
      {5B1B31A1-9562-11d2-8E3F-00A0C969723B}
```

### 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 3. |
| *SystemTable* | The image's EFI system table pointer. Type **EFI_SYSTEM_TABLE** is defined in Section 4.5.1. |
| *DeviceHandle* | The device handle that the EFI Image was loaded from. Type **EFI_HANDLE** is defined in Chapter 3. |
| *FilePath* | A pointer to the file path portion specific to *DeviceHandle* that the EFI Image was loaded from. Type **EFI_DEVICE_PATH** is defined in Chapter 3. |
| *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 3. |
| *ImageDataType* | The memory type that the data sections were loaded as. Type **EFI_MEMORY_TYPE** is defined in Chapter 3. |
| *Unload* | Function that unloads the image. See Section 4.1.1. |

## 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.

### 4.1.1 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 3. |

#### 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. |

## 4.2 EFI Image Header

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 IMAGE_SUBSYSTEM_EFI_APPLICATION          10
#define IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER  11
#define 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. Support for other machine code types are optional to the platform.

```
// PE32+ Machine type for EFI images
#define IMAGE_FILE_MACHINE_IA32       0x014c
#define IMAGE_FILE_MACHINE_IA64       0x0200
#define IMAGE_FILE_MACHINE_IBS        0xFC0D
```

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 "fixups" have been performed, control is transferred to a loaded image at the *AddressOfEntryPoint* reference according to the normal IA-32 or Itanium-based indirect calling conventions. All other linkage to and from an EFI image is done programmatically.

## 4.3 EFI Applications

Applications are loaded by the boot manager in the EFI firmware, or by other 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 **`Exit()`**, the application is unloaded from memory and control is returned to the shell that loaded the application.

When the boot manager loads an application, the image handle may be used to locate the "load options" for the application. The load options are those options that were stored in the **`LoadOptions`** field of the **`EFI_LOADED_IMAGE`** information for the application.

## 4.4 EFI OS Loaders

An EFI OS loader is a 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 device handles for access to devices that the firmware exposes. If the 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 **`DEVICE_IO`** 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 **`Exit()`** call with an error code and an *`ExitData`* that contains OS loader-specific data, including a Unicode string.

Once the OS loader successfully loads its operating system, it can take control of the system by using **`ExitBootServices()`**. After 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.

## 4.5 EFI Drivers

Drivers are loaded by the boot manager in the EFI firmware or by other applications. To load a 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 that are 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. If the driver returns from its entry point, or when it calls **`Exit()`** with an error code, the driver is unloaded from memory and control is returned to the shell that loaded the driver. If the driver returns **`EFI_SUCCESS`** from its entry point, it continues to reside in memory. If the driver is an **`EFIImageBootServiceDriver`**, the memory that the driver is loaded into is of type **`EfiBootServicesCode`** and **`EfiBootServicesData`**. Such memory regions revert back to normal memory when an OS loader exits boot services.

When the boot manager loads a driver, the image handle may be used to locate the "load options" for the driver. The load options are those options that were stored in the **LoadOptions** field of the **EFI_LOADED_IMAGE** information for the driver.

## 4.5.1 EFI Image Handoff State

Control is transferred to a loaded image at the *AddressOfEntryPoint* reference according to the normal indirect calling conventions for the image's *Machine* type. The entry point is a function of type **EFI_IMAGE_ENTRY_POINT**. All other linkage to and from an EFI image is done programmatically. See Chapter 3 for the full definition of **EFI_IMAGE_ENTRY_POINT**. Its prototype is repeated below, along with some additional comments.

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

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 non-zero 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 Section 4.1.

The following code shows the definition for the EFI system table. The EFI system table is provided as the second argument to a loaded image's entry point.

```
//
// EFI System Table
//

#define EFI_SYSTEM_TABLE_SIGNATURE      0x5453595320494249
#define EFI_SYSTEM_TABLE_REVISION       (1<<16) | (99)

typedef struct _EFI_SYSTEM_TABLE {
    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;

//
// Standard EFI table header
//

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

//
// EFI Configuration Table and GUID Declarations
//
#define MPS_TABLE_GUID \
    {0xeb9d2d2f, 0x2d88, 0x11d3, 0x9a, 0x16, 0x0, 0x90, 0x27, 0x3f, 0xc1, 0x4d}

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

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

#define SMBIOS_TABLE_GUID \
    {0xeb9d2d31, 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}
```

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

The EFI system table contains pointers to the runtime and boot services tables. The definitions for these tables are 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 Chapter 3. The **GetTime()** function is shown as an example.

## 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.*

```
// Example interface prototype

typedef
EFI_STATUS
(EFIAPI *EFI_GET_TIME) (
       OUT EFI_TIME                  *Time,
       OUT EFI_TIME_CAPABILITIES     *Capabilities OPTIONAL
       );

//
// EFI Runtime Services Table
//

#define EFI_RUNTIME_SERVICES_SIGNATURE  0x56524553544e5552
#define EFI_RUNTIME_SERVICES_REVISION   (1<<16) | (99)

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;


//
// EFI Boot Services Table
//

#define EFI_BOOT_SERVICES_SIGNATURE     0x56524553544f4f42
#define EFI_BOOT_SERVICES_REVISION      (1<<16) | (99)

typedef struct _EFI_BOOT_SERVICES {

    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
....// Note:  InstallConfigurationTable() is a "Miscellaneous Services"
....// function, but it was placed in this group to make use of a "reserved"
....// slot in the table.
    //
    EFI_INSTALL_PROTOCOL_INTERFACE   InstallProtocolInterface;
    EFI_REINSTALL_PROTOCOL_INTERFACE ReinstallProtocolInterface;
    EFI_UNINSTALL_PROTOCOL_INTERFACE UninstallProtocolInterface;
    EFI_HANDLE_PROTOCOL              HandleProtocol;
    EFI_HANDLE_PROTOCOL              PCHandleProtocol;
    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
....// See note about InstallConfigurationTable() under "Protocol Handler
....// Services" above.
    //

    EFI_GET_NEXT_MONOTONIC_COUNT     GetNextMonotonicCount;
    EFI_STALL                        Stall;
    EFI_SET_WATCHDOG_TIMER           SetWatchdogTimer;

} EFI_BOOT_SERVICES;
```

#### 4.5.1.1 IA-32 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 4-1 shows the stack after **`ImageEntryPoint`** has been called on IA-32 systems.

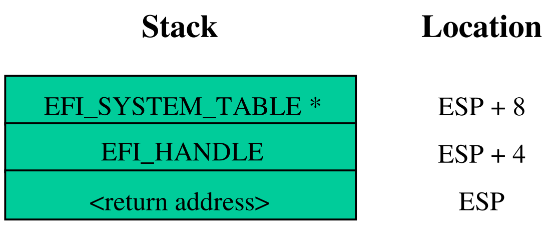

**Figure 4-1. Stack after ImageEntryPoint Called, IA-32**

### 4.5.1.2 Handoff State, Itanium-based Operating Systems

EFI uses the standard P64 C calling conventions that are defined for Itanium-based operating systems. Figure 4-2 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 header.

| Stack | Location | Register |
|---|---|---|
| EFI_SYSTEM_TABLE * | SP + 8 | out1 |
| EFI_HANDLE | SP | out0 |

**Figure 4-2. Stack after ImageEntryPoint Called, Itanium-based Systems**

The SAL specification (see "Related Information" in Chapter 1) 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.

# 5
# 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.

## 5.1 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.

## 5.2 EFI_DEVICE_PATH Protocol

This section provides a detailed description of the **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 Chapter 3. |

### 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.

## 5.3 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*.
- *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.

### 5.3.1 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 5-1 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 5.3; all other type and sub-type values are *Reserved*.

**Table 5-1. Generic Device Path Node Structure**

| Mnemonic | Byte Offset | Byte Length | Description |
|---|---|---|---|
| Type | 0 | 1 | Type 0x01 – Hardware Device Path<br>Type 0x02 – ACPI Device Path<br>Type 0x03 – Messaging Device Path<br>Type 0x04 – Media Device Path<br>Type 0x05 – BIOS Boot Specification Device Path<br>Type 0xFF – End of Hardware Device Path |
| Sub-Type | 1 | 1 | Sub-Type – Varies by Type. (See Table 5-2.) |
| 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 5-2):

- *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 **`EFI_HANDLE`** represents multiple devices. An example of this would be a handle that represents **`ConsoleOut`**, and consists of both a VGA console and serial output console. This handle would send the **`ConsoleOut`** stream 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 5-2. Device Path End Structure**

| Mnemonic | Byte Offset | Byte Length | Description |
|---|---|---|---|
| Type | 0 | 1 | Type 0x7F – End of Hardware Device Path<br>Type 0xFF – End of Hardware Device Path |
| Sub-Type | 1 | 1 | Sub-Type 0xFF – End Entire Device Path, or<br>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. |

## 5.3.2 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.

## 5.3.2.1 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 5-3. 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 |

## 5.3.2.2 PCCARD Device Path

**Table 5-4. 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. |
| Socket Number | 4 | 1 | Socket Number (0 = First Socket) |

### 5.3.2.3 Memory Mapped Device Path

**Table 5-5. 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** (See Chapter 3.) |
| Start Address | 8 | 8 | Starting Memory Address |
| End Address | 16 | 8 | Ending Memory Address |

### 5.3.2.4 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 5-6. 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. |

### 5.3.2.5 Controller Device Path

**Table 5-7. 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. |

## 5.3.3 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 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 and _UID device identification objects.

The _HID value is an optional device identification object that appears in the ACPI name space. The _HID must be used to describe any device that will be enumerated by the ACPI 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 is required to represent a PCI root bridge, since the PCI specification does not define the programming model for a PCI root bridge.

**Table 5-8. 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. |

## 5.3.4 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.

### 5.3.4.1 ATAPI Device Path

**Table 5-9. 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 |

### 5.3.4.2 SCSI Device Path

**Table 5-10. 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 |

### 5.3.4.3 Fibre Channel Device Path

**Table 5-11. 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 |

### 5.3.4.4 1394 Device Path

**Table 5-12. 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] |

[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.

### 5.3.4.5 USB Device Path

**Table 5-13. 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 Port Number | 4 | 1 | USB Port Number |
| End Point | 5 | 1 | USB Endpoint Number |

### 5.3.4.6 USB Class Device Path

**Table 5-14. 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. |

### 5.3.4.7 $I_2O$ Device Path

**Table 5-15. $I_2O$ 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 |

### 5.3.4.8 MAC Address Device Path

**Table 5-16. 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 |

### 5.3.4.9 IPv4 Device Path

**Table 5-17. 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<br>0x01 - The Source IP Address is statically bound |

### 5.3.4.10 IPv6 Device Path

**Table 5-18. 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<br>0x01 - The Source IP Address is statically bound |

### 5.3.4.11 InfiniBand† Device Path

**Table 5-19. 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 |
| Reserved | 4 | 4 | Reserved |
| Node GUID[1] | 8 | 8 | 64 bit node GUID[1] of the IOU |
| IOC GUID[1] | 16 | 8 | 64 bit GUID[1] of the IOC |
| Device ID | 24 | 8 | 64 bit persistent ID of the device |

[1] The usage of the term GUID is per the Infiniband specification. This is not the same as the **EFI_GUID** type defined in the EFI Specification.

### 5.3.4.12 UART Device Path

**Table 5-20. 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.<br>Parity 0x00 - Default Parity<br>Parity 0x01 - No Parity<br>Parity 0x02 - Even Parity<br>Parity 0x03 - Odd Parity<br>Parity 0x04 - Mark Parity<br>Parity 0x05 - Space Parity |
| Stop Bits | 18 | 1 | The number of stop bits for the UART style device.<br>Stop Bits 0x00 - Default Stop Bits<br>Stop Bits 0x01 - 1 Stop Bit<br>Stop Bits 0x02 - 1.5 Stop Bits<br>Stop Bits 0x03 - 2 Stop Bits |

#### 5.3.4.13 Vendor-Defined Messaging Device Path

**Table 5-21. 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 and VT-100 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 and VT-100 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 DEVICE_PATH_MESSAGING_PC_ANSI \
     { e0c14753-f9be-11d2-9a0c-0090273fc14d }

#define DEVICE_PATH_MESSAGING_VT_100  \
     { DFA66065-B419-11d3-9A2D-0090273FC14D }
```

### 5.3.5 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.

#### 5.3.5.1 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 5-22. 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)<br>0x01 – PC AT compatible MBR. Partition Start and Partition Size come from `PartitionStartingLBA` and `PartitionSizeInLBA` for the partition.<br>0x02 – EFI Partition Table Header. |
| Signature Type | 41 | 1 | Type of Disk Signature: (Unused values reserved)<br>0x00 – No Disk Signature.<br>0x01 – 32-bit signature from address 0x1b8 of the type 0x01 MBR.<br>0x02 – GUID signature. |

The following structure defines an 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;
```

### 5.3.5.2 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 5-23. 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. |

### 5.3.5.3 Vendor-Defined Media Device Path

**Table 5-24. 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. |

### 5.3.5.4 File Path Media Device Path

**Table 5-25. 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. |

### 5.3.5.5 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 5-26. 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. |

### 5.3.6 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 5-27. 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

## 5.4 Device Path Generation Rules

### 5.4.1 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.

## 5.4.2 Rules with ACPI _HID and _UID

As described in the ACPI specification, ACPI supports several different kinds of device identification objects, including _HID and _UID. EFI only supports _HID and _UID that are encoded in the 32-bit EISA-type ID format. The string format must not be used for _HID or _UID in the ACPI name space if that entry is to be correlated to an EFI Device Path. _UID 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 _UID field if the ACPI name space does not implement _UID. The _UID 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 non standard 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 following table maps ACPI _CRS devices to EFI Device Path.

**Table 5-28. 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 PNP0303, _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 and this will be stored in the ACPI Device Path _HID field. The _UID in the ACPI Device Path structure must match the _UID in the ACPI name space.

### 5.4.3 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.

The following table relates the ACPI _ADR bus definition to the EFI Device Path:

**Table 5-29. ACPI _ADR to EFI Device Path Mapping**

| ACPI _ADR Bus | EFI Device Path |
|---|---|
| EISA | *Not supported* |
| Floppy Bus | ACPI Device Path: _HID PNP0303, _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* |

### 5.4.4 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.

### 5.4.5 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.

### 5.4.6 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.

# 6
# 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.

## 6.1 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.

## 6.2 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 Section 6.2.1. |
| *Io* | Allows reads and writes to I/O space. See Section 6.2.1. |
| *Pci* | Allows reads and writes to PCI configuration space. See Section 6.2.1. |
| *Map* | Provides the device specific addresses needed to access system memory for DMA. |
| *PciDevicePath* | Provides an EFI Device Path for a PCI device with the given PCI configuration space address. |
| *Unmap* | Releases any resources allocated by Map. |
| *AllocateBuffer* | Allocates pages that are suitable for a common buffer mapping. |
| *Flush* | Flushes any posted write data to the device. |
| *FreeBuffer* | Free pages that were allocated with AllocateBuffer(). |

## 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()** 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 **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.

## 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;
```

### 6.2.1 DEVICE_IO.Mem(), .Io(), and .Pci()

#### Summary

Enable 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 6.2. |
| *Width* | Signifies the width of the I/O operations. Type **EFI_IO_WIDTH** is defined in Section 6.2. |
| *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 6-1. The caller must align the register number being accessed to be on a proper width boundary.

**Table 6-1. 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_UNSUPPORTED | The *Address* is not valid for this system. |
| EFI_INVALID_PARAMETER | *Width* or *Count*, or both, were invalid. |
| EFI_OUT_OF_RESOURCES | The request could not be completed due to a lack of resources. |

### 6.2.2 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**. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 6.2. |
| *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 6-1. |
| *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 3. |

#### 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 can not 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. |

## 6.2.3 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 6.2. |
| *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”. |
| *HostAddress* | The system memory address to map to the device. Type **EFI_ PHYSICAL_ADDRESS** is defined in Chapter 3. |
| *NumberOfBytes* | On input the number of bytes to map.<br>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 3. |
| *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 CPU and a bus master. The buffer is coherent from both the CPUs and the bus masters 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 can not 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`* can not 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. |

### 6.2.4 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 6.2. |
| *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. |

### 6.2.5 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**. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 6.2. |
| *Type* | The type allocation to perform. Type **EFI_ALLOCATE_TYPE** is defined in Chapter 3. |
| *MemoryType* | The type of memory to allocate, **EfiBootServicesData** or **EfiRuntimeServicesData**. Type **EFI_MEMORY_TYPE** is defined in Chapter 3. |
| *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 3. |

#### 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. |

### 6.2.6 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 6.2. |

#### 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. |

## 6.2.7 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**. Type **EFI_DEVICE_IO_INTERFACE** is defined in Section 6.2. |
| *Pages* | The number of pages to free. |
| *HostAddress* | The base address of the range to free. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 3. |

### 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 allocated. |
| EFI_INVALID_PARAMETER | The requested memory type is invalid. |

# 7
# 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).

## 7.1 Console I/O 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 Unicode ISO Latin 1 character set to enable standard terminal emulation software to be used with an EFI console. The ISO Latin 1 character set implements a superset of ASCII that has been extended to 16-bit characters. Any other number of Unicode code pages may be optionally supported.

## 7.2 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 7-1 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 7-1. 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 7-2.

**Table 7-2. 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. |

continued

**Table 7-2. EFI Scan Codes for SIMPLE_INPUT_INTERFACE** (continued)

| EFI Scan Code | Description |
|---|---|
| 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. |

## 7.3 SIMPLE_INPUT Protocol

### Summary

This protocol is used to obtain input from the *ConsoleIn* device.

### 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 Section 7.3.1. |
| *ReadKeyStroke* | Returns the next input character. See Section 7.3.2. |
| *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*.

### 7.3.1 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 7.3 |
| *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. |

### 7.3.2 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 7.3. |
| *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". |

#### 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 7-2. 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. |

## 7.4 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 7-1. 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 7-2.

## 7.5 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 Section 7.5.1. |
| *OutputString* | Displays the Unicode string on the device at the current cursor location. See Section 7.5.2. |
| *TestString* | Tests to see if the *ConsoleOut* device supports this Unicode string. See Section 7.5.3. |
| *QueryMode* | Queries information concerning the output device's supported text mode. See Section 7.5.4. |
| *SetMode* | Sets the current mode of the output device. See Section 7.5.5. |

| | |
|---|---|
| *SetAttribute* | Sets the foreground and background color of the text that is output. See Section 7.5.6. |
| *ClearScreen* | Clears the screen with the currently set background color. See Section 7.5.7. |
| *SetCursorPosition* | Sets the current cursor position. See Section 7.5.8. |
| *EnableCursor* | Turns the visibility of the cursor on/off. See Section 7.5.9. |
| *Mode* | Pointer to **SIMPLE_TEXT_OUTPUT_MODE** data. Type **SIMPLE_TEXT_OUTPUT_MODE** is defined in "Related Definitions". |

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 7-1. 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.

### 7.5.1 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 Section 7.5. |
| *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. |

## 7.5.2 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 Section 7.5. |
| *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 following 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. |

### 7.5.3 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 Section 7.5. |
| *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. |

### 7.5.4 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 Section 7.5. |
| *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. |

## 7.5.5 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 Section 7.5. |
| *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. |

### 7.5.6 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 Section 7.5. |
| *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". |

#### 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 mode information was returned. |
| 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. |

### 7.5.7 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 Section 7.5. |

#### 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. |

### 7.5.8 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 Section 7.5. |
| *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. |

### 7.5.9 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 Section 7.5. |
| *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 is not in a valid text mode, or the cursor position is invalid for the current mode. |

# 8
# Block I/O 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.

## 8.1 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". |
| *Reset* | Resets the block device hardware. See Section 8.1.1. |
| *ReadBlocks* | Reads the requested number of blocks from the device. See Section 8.1.2. |
| *WriteBlocks* | Writes the requested number of blocks to the device. See Section 8.1.3. |
| *FlushBlocks* | Flushes and cache blocks. This function is optional and only needs to be supported on block devices that cache writes. See Section 8.1.4. |

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 **WriteBlock()** 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 to read or write block(s). |
| *LastBlock* | The last logical block address on the device. If the media changes, then this field is updated. |

## 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;
```

## 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.

### 8.1.1 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 Section 8.1. |
| *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. |

### 8.1.2 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 Section 8.1. |
| *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 Section 8.1. |
| *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. |

### 8.1.3 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 Section 8.1. |
| *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 Section 8.1. |
| *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. |

### 8.1.4 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 Section 8.1. |

#### 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. |

# 9
# Disk I/O Protocol

This chapter 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.

## 9.1 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 Section 9.1.1. |
| *WriteDisk* | Writes data to the disk. See Section 9.1.2. |

## 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 a **`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 required formats that the firmware must automatically support are:

- 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.

### 9.1.1 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 Section 9.1. |
| *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. |

### 9.1.2 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 Section 9.1. |
| *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. |

# 10
# File System Protocol

This chapter 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.

## 10.1 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 Section 10.1.1. |

## 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 (see Section 10.2). 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

### 10.1.1 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. Type **EFI_FILE_IO_INTERFACE** is defined in Section 10.1. |
| *Root* | A pointer to the location to return the opened file handle for the root directory. Type **EFI_FILE** is defined in Section 10.2. |

#### 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 re-opened 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. |

## 10.2 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 Section 10.2.1. |
| *Close* | Closes the current file handle. See Section 10.2.2. |
| *Delete* | Deletes a file. See Section 10.2.3. |
| *Read* | Reads bytes from a file. See Section 10.2.4. |
| *Write* | Writes bytes to a file. See Section 10.2.5. |
| *GetPosition* | Returns the current file position. See Section 10.2.7. |
| *SetPosition* | Sets the current file position. See Section 10.2.6. |
| *GetInfo* | Gets the requested file or volume information. See Section 10.2.8. |
| *SetInfo* | Sets the requested file information. See Section 10.2.9. |
| *Flush* | Flushes all modified data associated with the file to the device. See Section 10.2.10. |

## 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.

## 10.2.1 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. Type **EFI_FILE** is defined in Section 10.2. |
| *NewHandle* | A pointer to the location to return the opened handle for the new file. Type **EFI_FILE** is defined in Section 10.2. |
| *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". |
| *Attributes* | Only valid for **EFI_FILE_MODE_CREATE,** in which case these are the attribute bits for the newly created file. See "Related Definitions". |

## 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 or if the file already exists, the operation fails.

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 re-opened.

## 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. |

### 10.2.2 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. Type **EFI_FILE** is defined in Section 10.2. |

#### 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. |

### 10.2.3 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. Type **EFI_FILE** is defined in Section 10.2. |

#### 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. |

### 10.2.4 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. Type **EFI_FILE** is defined in Section 10.2. |
| *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. See Section 10.2.11 for a discussion of **EFI_FILE_INFO**.

#### 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. |

## 10.2.5 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. Type **EFI_FILE** is defined in Section 10.2. |
| *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. |

### 10.2.6 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. Type **EFI_FILE** is defined in Section 10.2. |
| *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 –1, 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 –1 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 non-zero is not valid on open directories. |

### 10.2.7 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. Type **EFI_FILE** is defined in Section 10.2. |
| *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. |

### 10.2.8 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. Type **EFI_FILE** is defined in Section 10.2. |
| *InformationType* | The type identifier for the information being requested. Type **EFI_GUID** is defined in Chapter 3. See Section 10.2.11 and 10.2.12 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. |

## 10.2.9 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,
      OUT VOID                 *Buffer
      );
```

### Parameters

| | |
|---|---|
| *This* | A pointer to the **EFI_FILE** instance that is the file handle the information is for. Type **EFI_FILE** is defined in Section 10.2. |
| *InformationType* | The type identifier for the information being set. Type **EFI_GUID** is defined in Chapter 3. See Section 10.2.11 and10.2.12 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.* |

## 10.2.10 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. Type **EFI_FILE** is defined in Section 10.2. |

### 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. |

## 10.2.11 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". |
| *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).

## 10.2.12 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 files are typically grown by. |
| *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.

## 10.2.13 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
# Load File Protocol

This chapter 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.

## 11.1 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 Section 11.1.1. |

### 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.

### 11.1.1 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 3. |
| *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. |

# 12
# Serial I/O Protocol

This chapter defines the Serial I/O protocol. This protocol is used to abstract byte stream devices.

## 12.1 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* | Set the control bits on a serial device. These include Request to Send and Data Terminal Ready. |
| *GetControl* | Read the status of the control bits on a serial device. These include Clear to Send, Data Set Ready, Ring Indicator, and Carrier Detect. |
| *Write* | Send a buffer of characters to a serial device. |
| *Read* | Receive a buffer of characters from a serial device. |
| *Mode* | Pointer to **SERIAL_IO_MODE** data. Type **SERIAL_IO_MODE** is defined in "Related Definitions". |

## 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.

### 12.1.1 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 12.1. |

#### 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. |

### 12.1.2 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 12.1. |
| *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 Section 12.1. |
| *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 Section 12.1. |

## 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. |

### 12.1.3 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 12.1. |
| *Control* | Sets the bits of *Control* that are settable. See “Related Definitions”. |

#### 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. |

### 12.1.4 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 12.1. |
| *Control* | A pointer to return the current Control signals from the serial device. |

#### 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. |

### 12.1.5 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 12.1. |
| *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. |

### 12.1.6 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 12.1. |
| *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. |

# 13 Unicode Collation Protocol

This chapter 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.

## 13.1 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_STRLWR                      StrLwr;
      EFI_UNICODE_STRUPR                      StrUpr;
      EFI_UNICODE_FATTOSTR                    FatToStr;
      EFI_UNICODE_STRTOFAT                    StrToFat;
      CHAR8                                   *SupportedLanguages;
} UNICODE_COLLATION_INTERFACE;
```

### Parameters

| | |
|---|---|
| *StriColl* | Performs a case-insensitive comparison of two Null-terminated Unicode strings. See Section 13.1.1. |
| *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 Section 13.1.2. |
| *StrLwr* | Converts all the Unicode characters in a Null-terminated Unicode string to lower case Unicode characters. See Section 13.1.3. |

| | |
|---|---|
| *StrUpr* | Converts all the Unicode characters in a Null-terminated Unicode string to upper case Unicode characters. See Section 13.1.4 . |
| *FatToStr* | Converts an 8.3 FAT file name using an OEM character set to a Null-terminated Unicode string. See Section 13.1.5. |
| *StrToFat* | Converts a Null-terminated Unicode string to legal characters in a FAT filename using an OEM character set. See Section 13.1.6. |
| *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.

### 13.1.1 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** instance. Type **UNICODE_COLLATION_INTERFACE** is defined in Section 13.1. |
| *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. |

### 13.1.2 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_STRICOLL) (
      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 13.1. |
| *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>. |

**Examples patterns (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. |

## 13.1.3 UNICODE_COLLATION.StrLwr()

### Summary

Converts all the Unicode characters in a Null-terminated Unicode string to lower case 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 13.1. |
| *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 lower case equivalent if it has one. The converted string is returned in *String*.

### 13.1.4 UNICODE_COLLATION.StrUpr()

#### Summary

Converts all the Unicode characters in a Null-terminated Unicode string to upper case 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 13.1. |
| *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 upper case equivalent if it has one. The converted string is returned in *String*.

### 13.1.5 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 13.1. |
| *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.

### 13.1.6 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 13.1. |
| *String* | A pointer to a Null-terminated Unicode string. |
| *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. |

#### 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. |

# 14 PXE Base Code Protocol

This chapter 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.

## 14.1 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. |
| *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. |
| *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. |
| *Mtftp* | Performs TFTP and MTFTP services. |
| *UdpWrite* | Writes a UDP packet to the network interface. |
| *UdpRead* | Reads a UDP packet from the network interface. |
| *SetIpFilter* | Updates the IP receive filters of the network device. |
| *Arp* | Uses the ARP protocol to resolve a MAC address. |
| *SetParameters* | Updates the parameters that affect the operation of the PXE Base Code Protocol. |
| *SetStationIp* | Updates the station IP address and subnet mask values. |
| *SetPackets* | Updates the contents of the cached DHCP and Discover packets. |
| *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”. |

## 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. |
| *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. |
| *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;
```

```
//*******************************************************
// 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;

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

```
//*******************************************************
// 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;

//*******************************************************
// 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 2-1 PXE DHCP Options (Full List), in the PXE
// Specification.
//*******************************************************
```

**Table 14-1. 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)**<br>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.<br>**Type**<br>UNDI (1) = 0x01<br>**Versions**<br>WfM-1.1a 16-bit UNDI: MajorVer = 0x02. MinorVer = 0x00<br>PXE-2.0 16-bit UNDI: MajorVer = 0x02, MinorVer = 0x01<br>32/64-bit UNDI & H/W UNDI: MajorVer = 0x03, MinorVer = 0x00 |
| Client System Architecture | 93 [0x5D] | 2 [0x02] | **Type (2)**<br>Type is a two byte, network order, field that identifies the processor and programming environment of the client system.<br>**Types**<br>IA x86 PC = 0x00 0x00<br>IA64 EFI PC = 0x00 0x02<br>IA32 EFI PC = 0x00 0x06 |
| Class Identifier | 60 [0x3C] | 32 [0x20] | **"PXEClient:Arch:xxxxx:UNDI:yyyzzz"**<br>"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.)<br>xxxxx = ASCII represenetation of Client System Architecture.<br>yyyzzz = ASCII representation of Client Network Interface Identifier version numbers MajorVer(yyy) and MinorVer(zzz).<br>**Example**<br>"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 IA-32 or Itanium-based EFI environment. 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. The **`LOAD_FILE`** 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.

## 14.1.1 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 **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_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_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. |

## 14.1.2 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. |

### 14.1.3 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. |

### 14.1.4 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". |
| *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. |

### 14.1.5 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" 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 read-file and write-file operations, this is the size of the buffer specified by *BufferPtr*. For read file operations, if *BufferSize* is smaller than the size of the file being read, then this field will return the required size. For get-file size operations, this field 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" 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.

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. |

### 14.1.6 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. |

## 14.1.7 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 14-2. 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 14-3. 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 14-4. Source IP Filter Operation**

| OpFlags<br>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 14-5. Source UDP Port Filter Operation**

| OpFlags<br>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. |

### 14.1.8 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 IP's 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. |

### 14.1.9 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. |

## 14.1.10 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. |
| *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. |

## 14.1.11 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. |

### 14.1.12 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. |

## 14.2 EFI_PXE_BASE_CODE_CALLBACK Protocol

### Summary

This 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.

### 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. |

## 14.2.1 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 Simple Network Protocol

This chapter defines the Simple Network Protocol. This protocol provides a packet level interface to a network adapter.

## 15.1 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. |
| *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. |
| *Initialize* | Resets the network adapter and allocates the transmit and receive buffers. |
| *Reset* | Resets the network adapter and re-initializes it with the parameters provided in the previous call to *Initialize*. |
| *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. |
| *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. |
| *StationAddress* | Modifies or resets the current station address, if supported. |
| *Statistics* | Collects statistics from the network interface and allows the statistics to be reset. |
| *MCastIpToMac* | Maps a multicast IP address to a multicast HW MAC address. |
| *NvData* | Reads and writes the contents of the NVRAM devices attached to the network interface. |
| *GetStatus* | Reads the current interrupt status and the list of recycled transmit buffers from the network interface. |
| *Transmit* | Places a packet in the transmit queue. |
| *Receive* | Retrieves a packet from the receive queue, along with the status flags that describe the packet type. |
| *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”. |

## 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 thoseoperation, 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 can not 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 permenant 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.
//
// 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.

## 15.1.1 EFI_SIMPLE_NETWO RK.Start()

### Summary

Changes the state of a network interface from “stopped” “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. |

## 15.1.2 EFI_SIMPLE_NETWO RK.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. |

## 15.1.3 EFI_SIMPLE_NETWO RK.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. |

### 15.1.4 EFI_SIMPLE_NETWORK.Reset()

#### Summary

Resets a network adapter and re-initializes 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 re-initializes 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. |

## 15.1.5 EFI_SIMPLE_NETWO RK.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. |

## 15.1.6 EFI_SIMPLE_NETWO RK.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. |

## 15.1.7 EFI_SIMPLE_NETWO RK.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. |

## 15.1.8 EFI_SIMPLE_NETWO RK.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". |

### 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 than64 for ethernet. |
| *RxOversizeFrames* | Number of frames longer than the maxminum 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 mutlicast frames received and not dropped. |
| *RxCrcErrorFrames* | Number of frames w/ 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 maxminum 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 mutlicast frames transmitted and not dropped. |
| *TxCrcErrorFrames* | Number of frames w/ 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. |

## 15.1.9 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. |

## 15.1.10 EFI_SIMPLE_NETWO RK.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** in "Related Definitions" in Section 15.1.) |
| *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. |

### 15.1.11 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. |

## 15.1.12 EFI_SIMPLE_NETWO RK.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 non-zero, 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 non-zero, 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 non-zero 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 non-zero 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 non-zero 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 non-blocking 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. |

## 15.1.13 EFI_SIMPLE_NETWO RK.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

### Summary

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 G.

### 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 G. |
| | **H/W UNDI:** |
| | *Id* contains the address of the **!PXE** structure. |
| *ImageAddr* | Address of the un-relocated 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 un-relocated S/W UNDI image. |
| | **H/W UNDI:** |
| | *ImageAddr* contains zero. |
| *ImageSize* | Size of un-relocated 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 un-relocated 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”). |

| | |
|---|---|
| *MajorVer* | Major version number.<br>**16-bit UNDI:**<br>*MajorVer* comes from the third byte of the **UNDIRev** field in the **UNDI ROM ID** structure. Refer to the *Preboot Execution Environment (PXE) Specification*.<br>**32/64-bit S/W UNDI and H/W UNDI:**<br>*MajorVer* comes from the **Major** field in the **!PXE** structure. See Appendix G. |
| *MinorVer* | Minor version number.<br>**16-bit UNDI:**<br>*MinorVer* comes from the second byte of the **UNDIRev** field in the **UNDI ROM ID** structure. Refer to the *Preboot Execution Environment (PXE) Specification*.<br>**32/64-bit S/W UNDI and H/W UNDI:**<br>*MinorVer* comes from the **Minor** field in the **!PXE** structure. See Appendix G. |
| *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.

# 16
# 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 FAT-32 for a system partition, and FAT-12 or FAT-16 for removable media. The FAT-32 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.

## 16.1 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.

### 16.1.1 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 FAT-32). 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 FAT-32, FAT-16, and FAT-12 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.

### 16.1.2 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. The acronym LFN is used to denote long file names.

FAT 8.3 file names are always stored as upper case 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 can not exist in a single directory:

- “ThisIsAnExampleDirectory.Dir”
- “thisisanexamppledirectory.dir”
- THISISANEXAMPLEDIRECTORY.DIR
- ThisIsAnExampleDirectory.DIR

### 16.1.3 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 sub directories 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 sub directory name is up to the vendor, but all vendors must pick names that do not collide with any other vendor's sub directory 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 CPU architecture in each vendor sub directory. This guarantees that there is only one image that can be loaded from a vendor sub directory 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 sub directory 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 sub directories below the vendor sub directory. 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 sub directory below **EFI** called **BOOT**. There must only be one executable EFI image for each supported CPU architecture in the **BOOT** directory. For removable media to be bootable under EFI, it must be built in accordance with the rules layed 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
```

## 16.2 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 16-1.

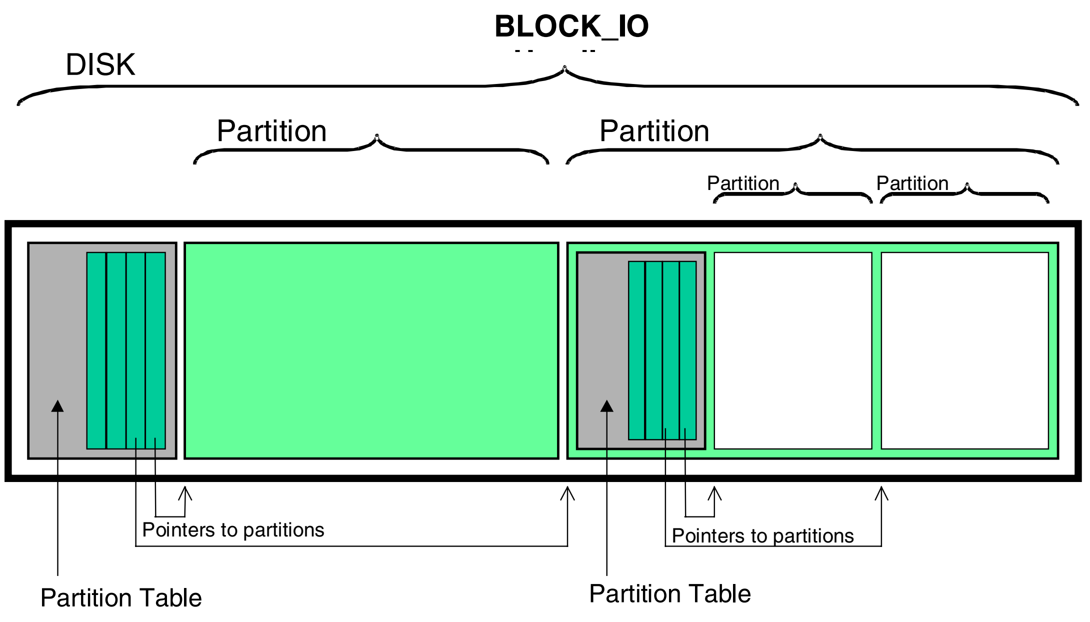

**Figure 16-1. Nesting of Legacy MBR Partition Records**

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).

## 16.2.1 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 16-2. The GUID Partition Table Header (see Table 16-1) 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 can not 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 LastUsableLBAon 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 it's 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.

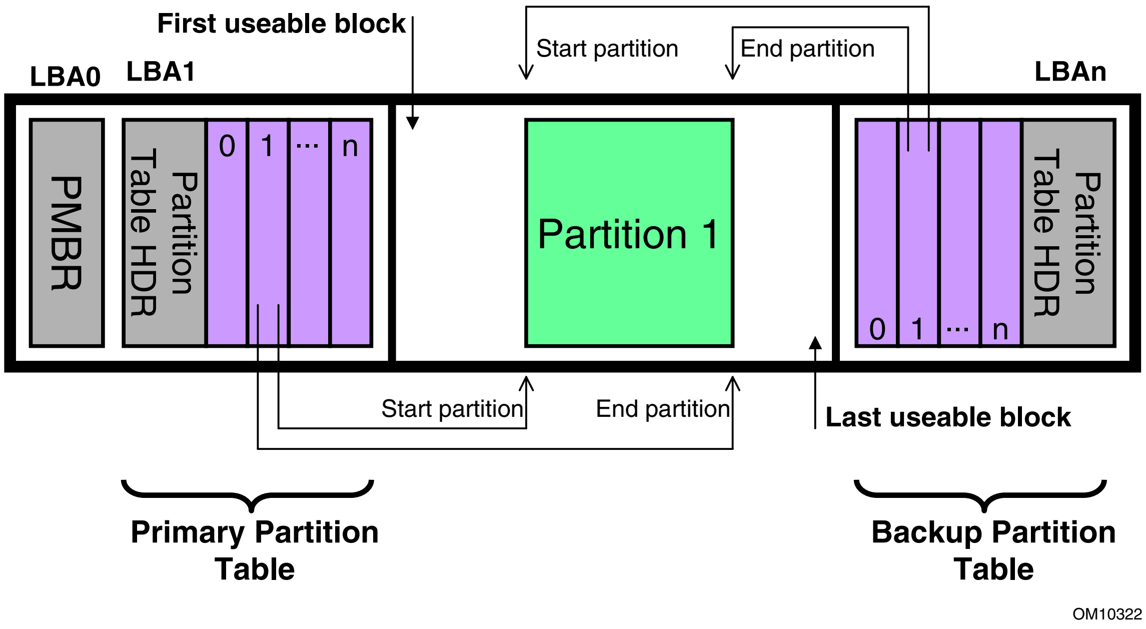

**Figure 16-2. GUID Partition Table (GPT) Scheme**

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 16-1. 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 ranged defined by HeaderSize is “check-summed”. |
| Reserved | 20 | 4 | Must be zero. |
| 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 it's 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 16-2. 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 16-3. 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 16-4. Defined GUID Partition Entry - Attributes**

| Bits | Description |
|---|---|
| Bit 0 | Required for the platform to function. The system can not 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. |

### 16.2.2 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.

### 16.2.3 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 16-5. 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 16-5. 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 Partiiton 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 16-6. 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. |

continued

**Table 16-6. Legacy Master Boot Record Partition Record** (continued)

| Mnemonic | Byte Offset | Byte Length | Description |
|---|---|---|---|
| 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.

## 16.2.4 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 16-7. 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 16-7. 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 non-bootable 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. |

## 16.3 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.

### 16.3.1 Removable Media

Removable media may contain a standard FAT-12, FAT-16, or FAT-32 file system. Legacy 1.44 MB floppy devices typically support a FAT-12 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 layed out in Section 17.4.1.1.

### 16.3.2 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 layed out in Section 17.4.1.1.

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 ½ 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†.

### 16.3.3 Hard Drive

Hard drives may contain multiple partitions as defined in Section 16.2 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 sub-directory as defined in Sections 16.1 and 16.2.

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.

### 16.3.4 CD-ROM and DVD-ROM

A CD-ROM or DVD-ROM may contain multiple partitions as defined Sections 16.1 and 16.2 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.

### 16.3.5 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.

# 17
# 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.

## 17.1 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 (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 325. 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 5-21 in Chapter 5). 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 16-2 in Chapter 16). 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 16-4 in Chapter 16). 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 buffer of variable length fields defined as follows:

## Descriptor

```
typedef struct {
        UINT32                              Attributes;
        UINT16                              FilePathListLength;
        CHAR16                              Description[];
        EFI_DEVICE_PATH                     FilePathList[];
        UINT8                               OptionalData[];
} EFI_LOAD_OPTION;
```

## 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** data structure. |
| *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 variable 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. |

## Related Definitions

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

## 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.

## 17.2 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 non-volatile. 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 pre-boot 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 pre-boot 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 17-1 lists the global variables.

**Table 17-1 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 non-volatile 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 can not 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 pre-selected 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.

## 17.3 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 non-existent 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 paragraph 17.4.1.1) 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.

## 17.4 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.

### 17.4.1 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 paragraph 17.4.1.1).

The format of the file system specified by EFI is contained in Chapter 16. 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.

#### 17.4.1.1 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 can not 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 17-2 lists the EFI image types.

**Table 17-2 EFI Image Types**

| Architecture | File name convention | PE Executable machine type * |
|---|---|---|
| IA-32 | BOOTIA32.EFI | 0x14c |
| Itanium-based 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.

## 17.4.2 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.

### 17.4.2.1 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_BC protocol (Chapter 14).

### 17.4.2.2 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.

# 18
# PCI Expansion ROM

The *PCI Local Bus Specification* defines how to discover expansion ROM code that comes from a ROM on a PCI Card. The expansion ROM can be executed to initialize a specific device or, possibly, to boot a system. PCI allows the ROM to contain several different images to accommodate different machine and processor architectures. This chapter explains how EFI images can be discovered and executed from a PCI expansion ROM. The EFI images are discovered using the basic methods outlined in the *PCI Local Bus Specification*, and then executed just like any other EFI image. The format and definition of an EFI image in a PCI expansion ROM are the same as the format and definition of an EFI image that is loaded from a disk or removable medium.

An EFI PCI expansion ROM can coexist with other image types in a single PCI ROM. The coexistence of multiple images in a PCI expansion ROM is detailed in the *PCI Local Bus Specification*. EFI utilizes a new PCI code type to define a platform-specific PCI Expansion ROM Header. The EFI expansion ROM header contains information about the image and a pointer to the start of the image.

## 18.1 Standard PCI Expansion ROM Header

All PCI expansion ROMs start with the standard header shown in Table 18-1. (The header is defined in the *PCI Local Bus Specification*, Revision 2.2). The table contains a simple signature, 0xAA55, and the offset to the PCI Data Structure. The Standard PCI Data Structure must be located within the first 64 KB of the ROM image and must be aligned on a four byte boundary.

**Table 18-1. 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 PCI Data Structure |

Table 18-2 defines the contents of the PCI Data Structure. (The definition is taken from the *PCI Local Bus Specification*, Revision 2.2). The code type field is used to identify the type of code contained in this section of the ROM. The following code types are assigned by the *PCI Local Bus Specification*, Revision 2.2:

- 0x00 – Intel® IA-32, PC-AT compatible
- 0x01 – Open Firmware standard for PCI
- 0x02 – Hewlett-Packard PA RISC
- 0x03-0xff – Reserved

EFI will coordinate with a future revision of the PCI specification to allocate the code type of 0x03 to represent EFI images. This code type will signify that EFI extensions are present in the standard PCI expansion ROM header.

**Table 18-2. PCI Data Structure**

| Offset | Byte Length | Description |
|---|---|---|
| 0x00 | 4 | Signature, the string 'PCIR' |
| 0x04 | 2 | Vendor Identification |
| 0x06 | 2 | Device Identification |
| 0x08 | 2 | Reserved |
| 0x0a | 2 | PCI Data Structure Length |
| 0x0c | 1 | PCI 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 |

## 18.2 EFI PCI Expansion ROM Header

A value of 0x03 in the code type field of the PCI data structure indicates that an EFI expansion ROM header is present in the system. The EFI PCI Expansion ROM Header contains all the standard entries defined in the *PCI Local Bus Specification*. It also contains the offset to the EFI driver image header. The offset to the EFI driver image header follows the same rules as the offset to the PCI data structure in the *PCI Local Bus Specification*. That is the EFI PCI Expansion ROM Header must be within the first 64 KB of the Standard PCI Expansion ROM header.

The EFI PCI Expansion ROM Header also contains information about the EFI driver image. The size of the image is given in units of 512 bytes. The maximum size of a PCI Expansion ROM is 16 MB. The initialization size includes the size of the EFI PCI expansion ROM header, the EFI image, and the PCI data structure. If the EFI PCI expansion ROM header is used in a context other than the *PCI Local Bus Specification* definition of an expansion ROM the image size may be set to zero.

The EFI expansion ROM header also contains some data that is included in the EFI image header. This data is a short cut, and allows the code parsing the EFI PCI expansion ROM header to know if it supports the image type that is pointed to by the EFI PCI expansion ROM header without decoding the image. These fields include the image signature, subsystem value, and machine type.

Table 18-3 defines the layout of an EFI PCI expansion ROM. Missing values will be supplied in a later version of the specification.

**Table 18-3. 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 | 10 | XX | Reserved |
| 0x16 | 2 | XX | Offset to EFI Image header |
| 0x18 | 2 | XX | Offset to PCI Data Structure |

## 18.3 Multiple Image Format Support

With the extension defined in this chapter it is possible to discover an EFI driver image in a PCI ROM. Since EFI images are defined with relocation there is no inherent limit as to where in memory it can be loaded. However, EFI driver images will only be loaded if enough free memory exists in the system.

An EFI system will only load an image if the platform supports the image type. Currently EFI has defined three image types: IA-32 image type; Itanium-based image type; and intermediate byte stream image type. The IA-32 and Itanium-based image type represent 32-bit and 64-bit native Intel architecture processor code that has knowledge about EFI interfaces. The intermediate byte stream type is a place holder for a new format that will be defined in a subsequent version of the EFI specification. A PCI expansion ROM may contain one or more EFI image types.

## 18.4 EFI PCI Expansion ROM Driver

PCI Expansion ROM drivers are no different from other EFI drivers that control hardware. (See Chapter 4 for details on how to construct an EFI driver.) To access a device, a driver needs to know the device's device path. For a driver loaded from a PCI Expansion ROM, the driver can examine the device path found in the **`LOADED_IMAGE`** structure for the driver image to obtain the device path of the device that driver image was loaded from. The driver must check that no other driver is already controlling the device.

For the driver to perform I/O and DMA operations with the device, the driver must use the proper **`DEVICE_IO`** protocol interfaces for the device. This is found by using the **`LocateDevicePath()`** function with the device path of the device and the ID of the **`DEVICE_IO`** protocol. See Chapter 6 for more information about the **`DEVICE_IO`** protocol.

# A
# GUID and Time Formats

All EFI GUIDs (Globally Unique Identifiers) have the format described in Appendix J of the *Wired for Management Baseline* S*pecification*. 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*. While this is the appendix for GUID it 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.

# 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, PCANSI, 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 PCANSI.

**Table B-1. EFI Scan Codes for SIMPLE_INPUT**

| EFI Scan Code | Description | ANSI X3.64 Codes | PCANSI 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 |
| 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 |

continued

**Table B-1. EFI Scan Codes for SIMPLE_INPUT** (continued)

| EFI Scan Code | Description | ANSI X3.64 Codes | PCANSI Codes | AT 101/102 Keyboard Scan Codes |
|---|---|---|---|---|
| 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 PCANSI or ANSI X3.64 terminals. Detailed descriptions of PCANSI 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**

| PCANSI 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. |
| 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. |

continued

**Table B-2. Control Sequences that Can Be Used to Implement SIMPLE_TEXT_OUTPUT** (continued)

| **PCANSI Codes** | **ANSI X3.64 Codes** | **Description** |
|---|---|---|
| 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. |

# 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.

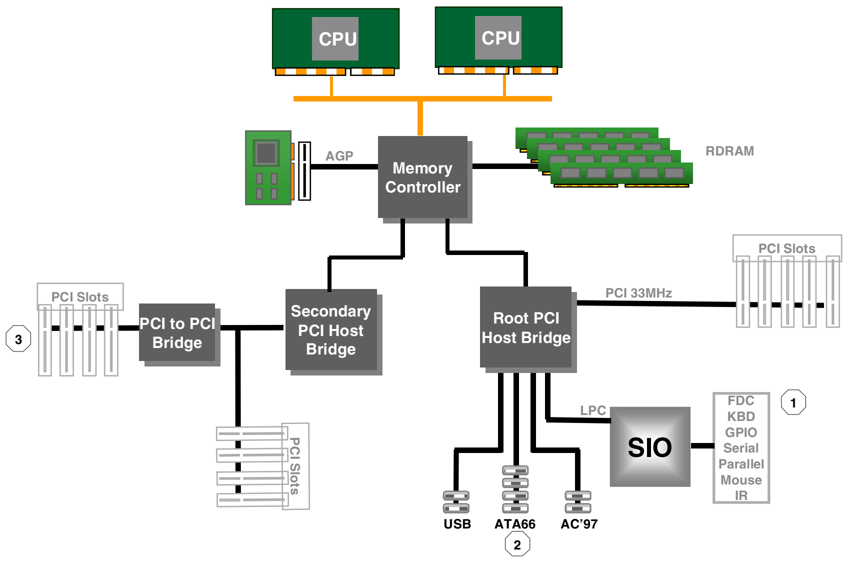

**Figure C-1. Example Computer System**

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*.

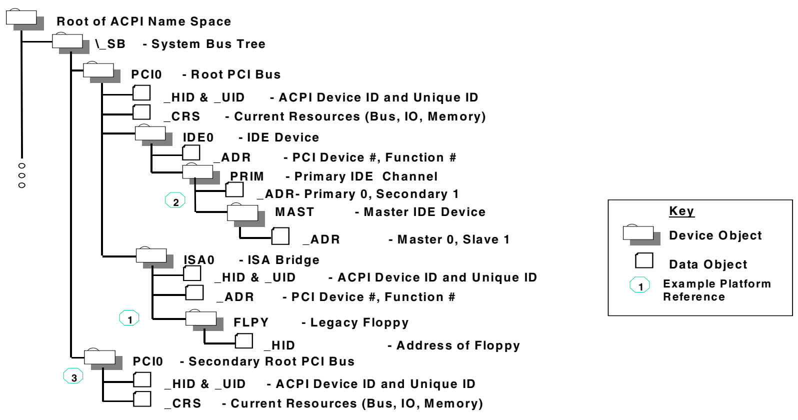
Root of ACPI Name Space
_SB - System Bus Tree
PCI0 - Root PCI Bus
_HID & _UID - ACPI Device ID and Unique ID
_CRS - Current Resources (Bus, IO, Memory)
IDE0 - IDE Device
_ADR - PCI Device #, Function #
PRIM - Primary IDE Channel
_ADR- Primary 0, Secondary 1
2
MAST - Master IDE Device
_ADR - Master 0, 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, IO, Memory)
Key
Device Object
Data Object
1 Example Platform Reference

**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 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 the that this device is the ATA master device on this primary ATA bus.

The EFI Device Path for the PCI IDE controller 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 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 non standard 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.

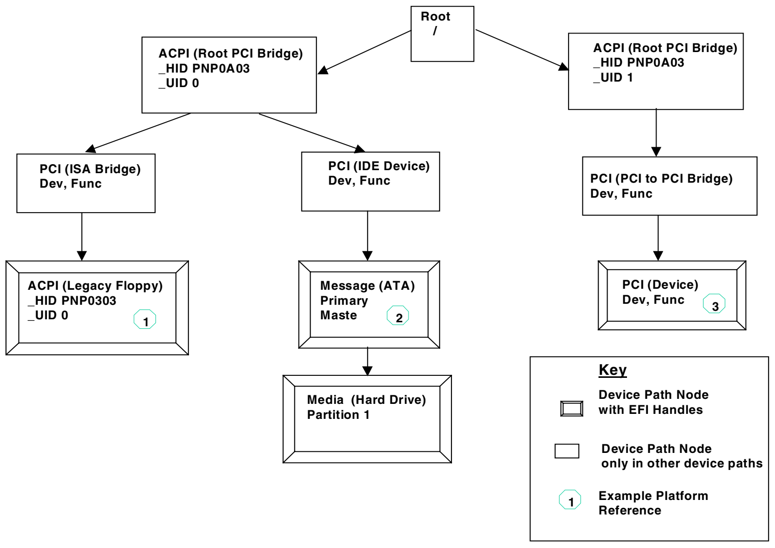

**Figure C-3. EFI Device Path Displayed As a Name Space**

# D
# Status Codes

EFI interfaces return an **EFI_STATUS** code. Tables D-2, D-3, and 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 | IA-64 Range | Description |
|---|---|---|
| `0x00000000-` `0x3fffffff` | `0x0000000000000000-` `0x3fffffffffffffff` | Success and warning codes reserved for use by EFI. See Tables D-2 and 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. |

**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. |

# E
# Alphabetic Function Lists

This appendix contains two tables that list all EFI functions alphabetically. Table E-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 E-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 E-1. Functions Listed in Alphabetic Order**

| Function Name | Service or Protocol | Sub-Service | Function Description |
|---|---|---|---|
| AllocateBuffer | Device 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. |
| CheckEvent | Boot Services | Event Services | Checks whether an event is in the signaled state. |
| 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. |
| ConvertPointer | Runtime Services | Virtual Memory Services | Converts internal pointers when switching to virtual addressing. |
| CreateEvent | Boot Services | Event Services | Creates a general-purpose event structure. |
| Delete | File System Protocol | | Deletes a file. |
| 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. |
| Discover | PXE Base Code Protocol | | Attempts to complete the PXE Boot Server and/or boot image discovery sequence. |

continued

**Table E-1. Functions Listed in Alphabetic Order** (continued)

| Function Name | Service or Protocol | Sub-Service | Function Description |
|---|---|---|---|
| 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. |
| EFI_IMAGE_ENTRY_POINT | Boot Services | Image Services | Prototype of an EFI Image's entry point. |
| 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. |
| 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. |
| FlushBlocks | Block I/O Protocol | | Flushes any cached blocks. |
| FreeBuffer | Device I/O Protocol | | Frees pages that were allocated with AllocateBuffer. |
| FreePages | Boot Services | Memory Allocation Services | Frees memory pages. |
| FreePool | Boot Services | Memory Allocation Services | Frees allocated pool. |
| GetControl | Serial I/O Protocol | | Reads the status of the control bits on a serial device. |
| GetInfo | File System Protocol | | Gets the requested file or volume information. |
| GetMemoryMap | Boot Services | Memory Allocation Services | Returns the current boot services memory map and memory map key. |
| 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. |
| GetStatus | Simple Network Protocol | | Reads the current interrupt status and recycled transmit buffer status from the network interface. |

continued

**Table E-1. Functions Listed in Alphabetic Order** (continued)

| Function Name | Service or Protocol | Sub-Service | Function Description |
|---|---|---|---|
| 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 | 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 |
| InstallConfigurationTable | Boot Services | Miscellaneous Services | Adds, updates, or removes a configuration table from the EFI System Table. |
| InstallProtocolInterface | Boot Services | Protocol Handler Services | Adds a protocol interface to an existing or new device handle. |
| Io.Read | Device I/O Protocol | | Reads from I/O ports on a bus. |
| Io.Write | Device I/O Protocol | | Writes to I/O ports on a bus. |
| 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. |
| Map | Device I/O Protocol | | Provides the device specific addresses needed to access host memory for DMA. |
| 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.Write | Device I/O Protocol | | Writes to memory on a bus. |

continued

**Table E-1. Functions Listed in Alphabetic Order** (continued)

| Function Name | Service or Protocol | Sub-Service | Function Description |
|---|---|---|---|
| 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. |
| NVData | Simple Network Protocol | | Allows read and writes to the NVRAM device attached to a network interface. |
| Open | File System Protocol | | Opens or creates a new file. |
| OpenVolume | Simple File System Protocol | | Opens the volume for file I/O access. |
| OutputString | Simple Text Output Protocol | | Displays the Unicode string on the device at the current cursor location. |
| Pci.Read | Device I/O Protocol | | Reads from PCI Configuration Space. |
| Pci.Write | Device I/O Protocol | | 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. |
| 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 | 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. |
| RegisterProtocolNotify | Boot Services | Protocol Handler Services | Registers for protocol interface installation notifications |

continued

**Table E-1. Functions Listed in Alphabetic Order** (continued)

| Function Name | Service or Protocol | Sub-Service | Function Description |
|---|---|---|---|
| ReinstallProtocolInterface | Boot Services | Protocol Handler Services | Replaces a protocol interface. |
| Reset | Block I/O Protocol | | Resets the block device 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 re-initializes it with the parameters that were provided in the previous call to Initialize(). |
| Reset | Simple Text Output Protocol | | Reset the ConsoleOut device. |
| ResetSystem | Runtime Services | Miscellaneous Services | Resets the entire platform. |
| RestoreTPL | Boot Services | Task Priority Services | Restores/lowers the task priority level. |
| SetAttribute | Simple Text Output Protocol | | Sets the foreground and background color of the text that is output. |
| SetAttributes | Serial I/O Protocol | | Sets communication parameters for a serial device. |
| 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. |
| SetMode | Simple Text Output Protocol | | Sets the current mode of the output device. |
| 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. |
| SetStationIp | PXE Base Code Protocol | | Updates the station IP address and/or subnet mask values. |

continued

**Table E-1. Functions Listed in Alphabetic Order** (continued)

| **Function Name** | **Service or Protocol** | **Sub-Service** | **Function Description** |
|---|---|---|---|
| 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 | Simple Network Protocol | | Resets the network adapter and leaves it in a state safe for another driver to initialize. |
| SignalEvent | Boot Services | Event Services | Signals an event. |
| Stall | Boot Services | Miscellaneous Services | Stalls the processor. |
| 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. |
| StartImage | Boot Services | Image Services | Function to transfer control to the Image's entry point. |
| 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. |
| 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. |
| StriColl | Unicode Collation Protocol | | Performs a case-insensitive comparison between two Unicode strings. |

continued

**Table E-1. Functions Listed in Alphabetic Order** (continued)

| Function Name | Service or Protocol | Sub-Service | Function Description |
|---|---|---|---|
| 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. |
| 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. |
| 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. |
| UninstallProtocolInterface | Boot Services | Protocol Handler Services | Removes a protocol interface from a device handle. |
| Unload | Loaded Image | | Requests an image to unload. |
| UnloadImage | Boot Services | Image Services | Unloads an image. |
| Unmap | Device I/O Protocol | | Releases any resources allocated by Map(). |
| WaitForEvent | Boot Services | Event Services | Stops execution until an event is signaled. |
| 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 E-2. Functions Listed Alphabetically Within 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 Services | AllocatePages | Allocates memory pages of a particular type. |
| | AllocatePool | Allocates pool of a particular type. |
| | CheckEvent | Checks whether an event is in the signaled state. |
| | CloseEvent | Closes and frees an event structure. |
| | CreateEvent | Creates a general-purpose event structure. |
| | 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 |
| | 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. |
| | 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. |
| | SetTimer | Sets an event to be signaled at a particular time. |
| | SetWatchdogTimer | Resets and sets the system's watchdog timer. |
| | SignalEvent | Signals an event. |

continued

**Table E-2. Functions Listed Alphabetically Within Service or Protocol** (continued)

| Service or Protocol | Function | Function Description |
|---|---|---|
| Boot Services (cont.) | Stall | Stalls the processor. |
| | StartImage | Function to transfer control to the Image's entry point. |
| | UninstallProtocolInterface | Removes a protocol interface from a device handle. |
| | UnloadImage | Unloads an image. |
| | WaitForEvent | Stops execution until an event is signaled. |
| 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. |
| 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. |

continued

**Table E-2. Functions Listed Alphabetically Within 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 E-2. Functions Listed Alphabetically Within Service or Protocol** (continued)

| **Service or Protocol** | **Function** | **Function Description** |
|---|---|---|
| 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. |
| | Reset | Resets the network adapter, and re-initializes it with the parameters that were provided in the previous call to Initialize(). |
| | 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. |

continued

**Table E-2. Functions Listed Alphabetically Within Service or Protocol** (continued)

| Service or Protocol | Function | Function Description |
|---|---|---|
| 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 | Reset 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. |
| 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. |

# F 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 non standard 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**

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.

**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.

**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**

Basic Input/Output System. A collection of low-level I/O service routines.

**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.

**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_IO 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**

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.

**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 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.

**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**.

**Coherency Domain**

The global set of resources that is visible to at least one processor in a platform.

**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**

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**.

**Device Path Instance**

When an **EFI Handle** represents multiple devices, it is possible for a device path to contain multiple device paths. An example of this would be a handle that represents **ConsoleOut** and consists of both a VGA console and a serial output console. The handle 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, sub-type, 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**.

**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.

**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-compliant**

Refers to a platform that complies with this specification.

**EFI-conformant**

See EFI-compliant.

**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.

**EM** (Enhanced Mode)

The 64-bit architecture extension that makes up part of the Intel Itanium architecture.

**End of Hardware Device Path**

A **Device Path** which, depending on the sub-type, is used to indicate the end of the Device Path instance or Device Path structure.

**Event**

An EFI data structure that describes an "event" — for example, the expiration of a timer.

**Event Services**

The set of functions used to manage events. Includes `CheckEvent()`, `CreateEvent()`, `CloseEvent()`, `SignalEvent()`, and `WaitForEvent()`.

**FAT**

See **File Allocation Table**.

**FAT File System**

The file system on which the EFI file system is based. See **File Allocation Table** and **System Partition**.

**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: FAT-12, which uses 12 bits for cluster numbers; FAT-16, which uses 16 bits; and Fat-32, 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).

**GUID** (Globally Unique Identifier)

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**

A contiguous group of sectors on an **EFI Hard Disk**.

**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 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 Entry**

A data structure that characterizes a **GUID Partition**. Among other things, it specifies the starting and ending LBA of the partition.

**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**

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.

**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`.

**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, and Pentium III processors.

**Intel Itanium Architecture**

A new 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 *IA-64 Architecture Software Developer's Manual*.

**Intel Architecture Platform Architecture**

A collective term for PC-AT-class computers and other systems based on Intel Architecture processors of all families.

**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**

A file containing executable code. When started, a loaded image is given its image handle and can use it to obtain relevant image data.

**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**.

**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** (TFTP)

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 EFI a **Device Path**.

**NBP**

See **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**.

**Page Memory**

A set of contiguous pages. Page memory is allocated by **`AllocatePages()`** and returned by **`FreePages()`**.

**Partition**

See **System Partition**.

**Partition Discovery**

The process of scanning a block device to determine whether it contains a **Partition**.

**PC-AT**

Refers to a PC platform that uses the AT form factor for their motherboards.

**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.

**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.

**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 Handler Services**

The set of functions used to manipulate handles, protocols, and protocol interfaces. Includes `InstallProtocolInterface()`, `UninstallProtocolInterface()`, `ReInstallProtocolInterface()`, `HandleProtocol()`, `RegisterProtocolNotify()`, `LocateHandle()`,and `LocateDevicePath()`.

**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**.

**PXE**

See **Preboot Execution Environment**.

**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**.

**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.

**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**.

**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.

**Unicode**

An industry standard internationalized character set used for human readable message display.

**Unicode Collation Protocol**

A protocol that is used during boot services to perform case-insensitive comparisons of Unicode strings.

**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**

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()`.

**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**

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.

# G
# 32/64-Bit UNDI Specification

## G.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.*

### G.1.1 Definitions

**Table G-1. Definitions**

| Term | Definition |
|---|---|
| BC | **BaseCode**<br>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**<br>This is a network device that is built onto the motherboard (or baseboard) of the machine. |
| NBP | **Network Bootstrap Program**<br>This is the first program that is downloaded into a machine that has selected a PXE capable device for remote boot services.<br>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.<br>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**<br>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.)). |
| ROM | **Read-Only Memory**<br>When used in this specification, ROM refers to a non-volatile memory storage device on a NIC. |

continued

**Table G-1. Definitions** (continued)

| Term | Definition |
|---|---|
| PXE | **Preboot Execution Environment**<br>The complete PXE specification covers three areas; the client, the network and the server.<br>**Client**<br>• Makes network devices into bootable devices.<br>• Provides APIs for PXE protocol modules in EFI and for universal drivers in the OS.<br>**Network**<br>• Uses existing technology: DHCP, TFTP, etc.<br>• Adds "vendor specific" tags to DHCP to define PXE specific operation within DHCP.<br>• Adds multicast TFTP for high bandwidth remote boot applications.<br>• Defines Bootserver discovery based on DHCP packet format.<br>**Server**<br>• **Bootserver:** Responds to Bootserver discovery requests and serves up remote boot images.<br>• **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.<br>• **MTFTP:** Adds multicast support to a TFTP server.<br>• **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**<br>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. |

## G.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. The table below includes all of the specifications referenced in this document.

**Table G-2. Referenced Specification**

| 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. |

continued

**Table G-2. Referenced Specification** (continued)

| Acronym | Protocol/Specification |
|---|---|
| BOOTP | **Bootstrap Protocol** – http://www.ietf.org/rfc/rfc0951.txt - This reference is included for backward compatibility. BC protocol supports DHCP and BOOTP.<br>Required reading for those implementing the BC protocol or PXE Bootservers. |
| DHCP | **Dynamic Host Configuration Protocol**<br>DHCP for Ipv4 (protocol: http://www.ietf.org/rfc/rfc2131.txt, options: http://www.ietf.org/rfc/rfc2132.txt)<br>Required reading for those implementing the BC protocol or PXE Bootservers. |
| EFI | **Extensible Firmware Interface** – http://developer.intel.com/technology/efi/index.htm<br>Required reading for those implementing NBPs, OS loaders and preboot applications for machines with the EFI preboot environment. |
| ICMP | **Internet Control Message Protocol**<br>ICMP for Ipv4: http://www.ietf.org/rfc/rfc0792.txt<br>ICMP for Ipv6: http://www.ietf.org/rfc/rfc2463.txt<br>Required reading for those implementing the BC protocol. |
| IETF | **Internet Engineering Task Force** – http://www.ietf.org/<br>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<br>Required reading for those implementing the BC protocol. |
| IP | **Internet Protocol**<br>Ipv4: http://www.ietf.org/rfc/rfc0791.txt<br>Ipv6: http://www.ietf.org/rfc/rfc2460.txt & http://www.ipv6.org<br>Required reading for those implementing the BC protocol. |
| MTFTP | **Multicast TFTP** – Defined in the 16-bit PXE specification.<br>Required reading for those implementing the BC protocol. |
| PCI | **Peripheral Component Interface** – http://www.pcisig.com/ - Source for PCI specifications.<br>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/techs/specs.html<br>Source for PnP specifications. |
| PXE | **Preboot eXecution Environment**<br>16-bit PXE v2.1: ftp://download.intel.com/ial/wfm/pxespec.pdf<br>Required reading. |
| RFC | **Request For Comments** – http://www.ietf.org/rfc.html |

continued

Table G-2. **Referenced Specification** (continued)

| Acronym | Protocol/Specification |
|---|---|
| TCP | **Transmission Control Protocol**<br>TCPv4: http://www.ietf.org/rfc/rfc0793.txt<br>TCPv6: ftp://ftp.ipv6.org/pub/rfc/rfc2147.txt<br>Required reading for those implementing the BC protocol. |
| TFTP | **Trivial File Transfer Protocol**<br>TFTP over IPv4 (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).<br>TFTP over IPv6: %%TBD need URL and an RFC!<br>Required reading for those implementing the BC protocol. |
| UDP | **User Datagram Protocol**<br>UDP over IPv4: http://www.ietf.org/rfc/rfc0768.txt<br>UDP over IPv6: http://www.ietf.org/rfc/rfc2454.txt<br>Required reading for those implementing the BC protocol. |
| WfM | **Wired for Management**<br>ftp://download.intel.com/ial/wfm/baseline.pdf<br>Recommended reading for those implementing the BC protocol or PXE Bootservers. |

## G.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. The figure below depicts an application bound to an OS protocol stack, which is in turn bound to a protocol driver that is bound to three NICs. The table below the figure gives a brief list of pros and cons about each type of driver implementation.

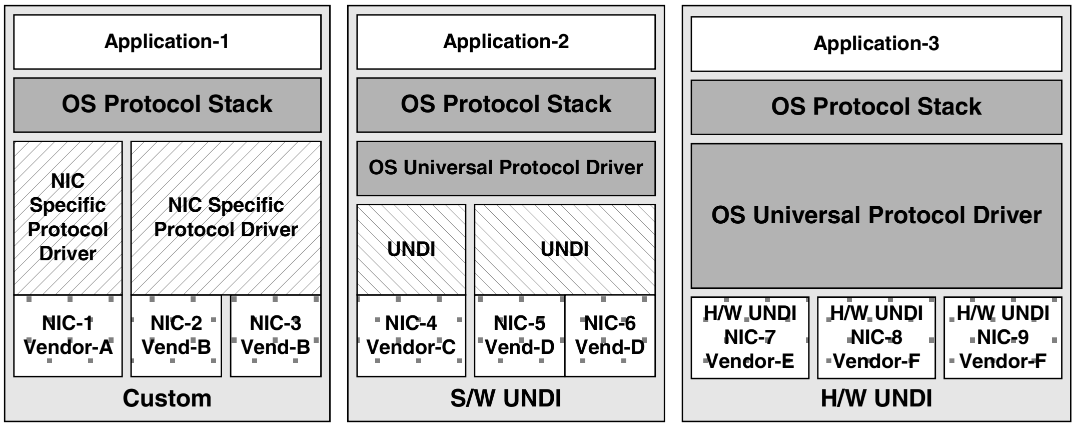

**Figure G-1. Network Stacks with Three Classes of Drivers**

**Table G-3. Driver Types: Pros and Cons**

| Driver | Pro | Con |
|---|---|---|
| Custom | • Can be very fast and efficient. NIC vendor tunes driver to OS & device.<br>• OS vendor does not have to write NIC driver. | • New driver for each OS/architecture must be maintained by NIC vendor.<br>• OS vendor must trust code supplied by third-party.<br>• OS vendor cannot test all possible driver/NIC versions.<br>• Driver must be installed before NIC can be used.<br>• Possible performance sink if driver is poorly written.<br>• 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.<br>• OS vendor can tune the universal protocol driver for best OS performance.<br>• NIC vendor only has to write one driver per CPU architecture. | • Slightly slower than Custom or H/W UNDI because of extra call layer between protocol stack and NIC.<br>• S/W UNDI driver must be loaded before NIC can be used.<br>• OS vendor has to write the universal driver.<br>• Possible performance sink if driver is poorly written.<br>• Possible security risk if driver has back door. |
| H/W UNDI | • H/W UNDI provides a common architectural interface to all network devices.<br>• OS vendor controls all security and performance issues in network stack.<br>• NIC vendor does not have to write any drivers.<br>• 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). |

## G.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.

### G.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). 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 | Implemenation | | | |
| 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 | Implemenation | | | |
| 0x10 | EntryPoint | | | |
| 0x14 | | | | |
| 0x18 | reserved | | | #bus |
| 0x1C | BusType(s) | | | |
| 0x20 | More BusType(s) | | | |

**Figure G-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 I/O, space is architecture dependent and is outside the scope of this specification.

The !PXE structure for S/W UNDI can be loaded into system memory from one of three places; ROM on a NIC, system non-volatile 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 the table below.

**Table G-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.<br>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 G-4. !PXE Structure Field Definitions** (continued)

| Identifier | Value | Description |
|---|---|---|
| Implementation | Varies | Identifies type of UNDI<br>(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).<br>Bit 0x00: Command completion interrupts supported (1) or not supported (0)<br>Bit 0x01: Packet received interrupts supported (1) or not supported (0)<br>Bit 0x02: Transmit complete interrupts supported (1) or not supported (0)<br>Bit 0x03: Software interrupt supported (1) or not supported (0)<br>Bit 0x04: Filtered multicast receives supported (1) or not supported (0)<br>Bit 0x05: Broadcast receives supported (1) or not supported (0)<br>Bit 0x06: Promiscuous receives supported (1) or not supported (0)<br>Bit 0x07: Promiscuous multicast receives supported (1) or not supported (0)<br>Bit 0x08: Station MAC address settable (1) or not settable (0)<br>Bit 0x09: Statistics supported (1) or not supported (0)<br>Bit 0x0A,0x0B: NvData not available (0), read only (1), sparse write supported (2), bulk write supported (3)<br>Bit 0x0C: Multiple frames per command supported (1) or not supported (0)<br>Bit 0x0D: Command queuing supported (1) or not supported (0)<br>Bit 0x0E: Command linking supported (1) or not supported (0)<br>Bit 0x0F: Packet fragmenting supported (1) or not supported (0)<br>Bit 0x10: Device can address 64 bits (1) or only 32 bits (0)<br>Bit 0x1E: S/W UNDI: Entry point is virtual address (1) or unsigned offset from start of !PXE structure (0).<br>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.<br>**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 G-4. !PXE Structure Field Definitions** (continued)

| Identifier | Value | Description |
|---|---|---|
| Status | Varies | UNDI operation, command and interrupt status flags.<br>This is a read-only port. Undefined status bits must be set to zero. Reading this port does NOT clear the status.<br>Bit 0x00**:** Command completion interrupt pending (1) or not pending (0)<br>Bit 0x01**:** Packet received interrupt pending (1) or not pending (0)<br>Bit 0x02**:** Transmit complete interrupt pending (1) or not pending (0)<br>Bit 0x03**:** Software interrupt pending (1) or not pending (0)<br>Bit 0x04**:** Command completion interrupts enabled (1) or disabled (0)<br>Bit 0x05**:** Packet receive interrupts enabled (1) or disabled (0)<br>Bit 0x06**:** Transmit complete interrupts enabled (1) or disabled (0)<br>Bit 0x07**:** Software interrupts enabled (1) or disabled (0)<br>Bit 0x08**:** Unicast receive enabled (1) or disabled (0)<br>Bit 0x09**:** Filtered multicast receive enabled (1) or disabled (0)<br>Bit 0x0A**:** Broadcast receive enabled (1) or disabled (0)<br>Bit 0x0B**:** Promiscuous receive enabled (1) or disabled (0)<br>Bit 0x0C**:** Promiscuous multicast receive enabled (1) or disabled (0)<br>Bit 0x1D**:** Command failed (1) or command succeeded (0)<br>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.<br>Bit 0x00: Clear command completion interrupt (1) or NOP (0)<br>Bit 0x01: Clear packet received interrupt (1) or NOP (0)<br>Bit 0x02: Clear transmit complete interrupt (1) or NOP (0)<br>Bit 0x03: Clear software interrupt (1) or NOP (0)<br>Bit 0x04: Command completion interrupt enable (1) or disable (0)<br>Bit 0x05: Packet receive interrupt enable (1) or disable (0)<br>Bit 0x06: Transmit complete interrupt enable (1) or disable (0)<br>Bit 0x07: Software interrupt enable (1) or disable (0). Setting this bit to (1) also generates a software interrupt.<br>Bit 0x08: Unicast receive enable (1) or disable (0)<br>Bit 0x09: Filtered multicast receive enable (1) or disable (0)<br>Bit 0x0A: Broadcast receive enable (1) or disable (0)<br>Bit 0x0B: Promiscuous receive enable (1) or disable (0)<br>Bit 0x0C: Promiscuous multicast receive enable (1) or disable (0)<br>Bit 0x1F: Operation type: Clear interrupt and/or filter (0), Issue command (1) |
| CDBaddr | Varies | Write the physical address of a CDB to this port. (Done with one 64-bit or two 32-bit writes, depending on CPU 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. |

continued

**Table G-4. !PXE Structure Field Definitions** (continued)

| **Identifier** | **Value** | **Description** |
|---|---|---|
| **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 four 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. |

### G.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 below is a high level diagram on how commands are written to both H/W and S/W UNDI.

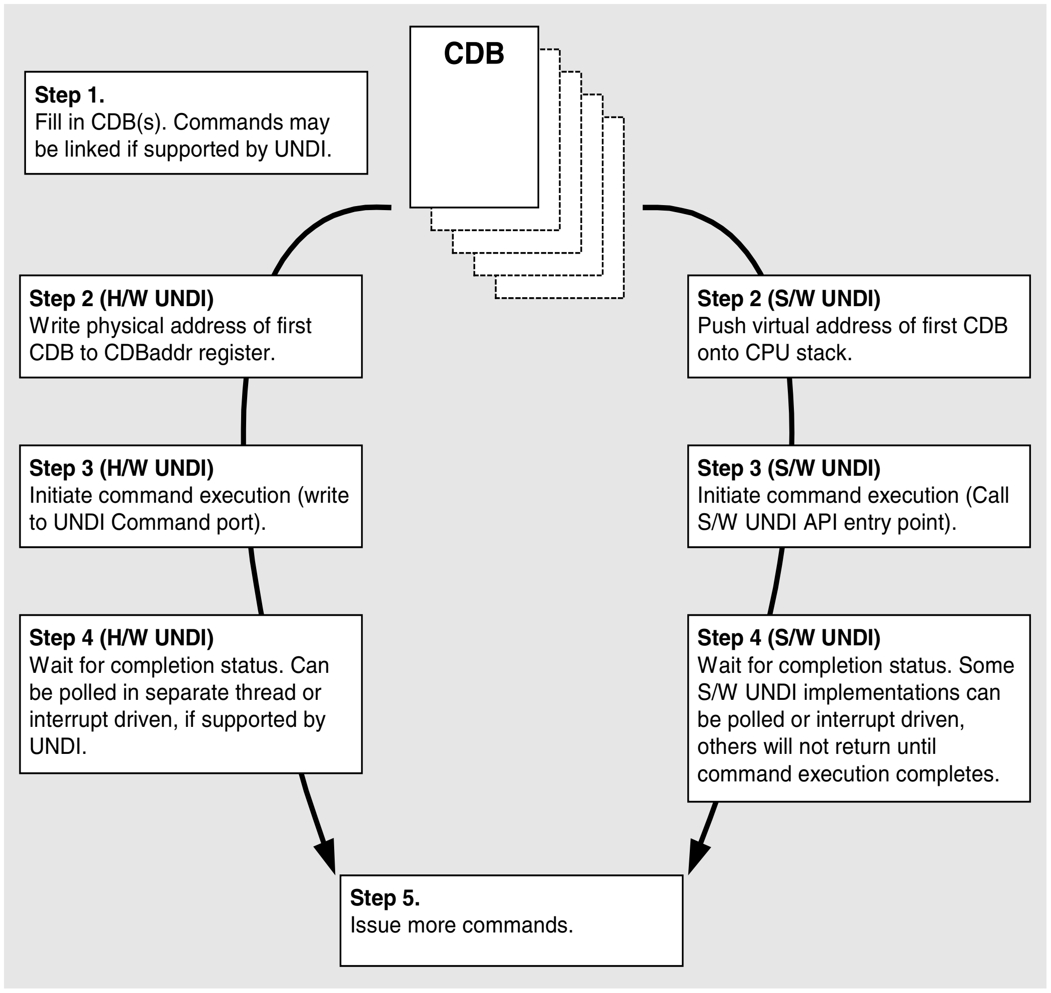
CDB
Step 1.
Fill in CDB(s). Commands may be linked if supported by UNDI.
Step 2 (H/W UNDI)
Write physical address of first CDB to CDBaddr register.
Step 3 (H/W UNDI)
Initiate command execution (write to UNDI Command port).
Step 4 (H/W UNDI)
Wait for completion status. Can be polled in separate thread or 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.

**Figure G-3. Issuing UNDI Commands**

## G.2.2 UNDI Command Format

The format of the CDB is the same for all UNDI commands. 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.

**CDB**
Command Descriptor Block

| Offset | 0x00 | 0x01 | 0x02 | 0x03 |
|---|---|---|---|---|
| 0x00 | **OpCode** | | **OpFlags** | |
| 0x04 | **CPBsize** | | **DBsize** | |
| 0x08 | **CPBaddr** | | | |
| 0x0C | | | | |
| 0x10 | **DBaddr** | | | |
| 0x14 | | | | |
| 0x18 | **StatCode** | | **StatFlags** | |
| 0x1C | **IFnum** | | **Control** | |

**Figure G-4. UNDI Command Descriptor Block (CDB)**

Descriptions of the CDB fields are given in the table below.

**Table G-5. UNDI CDB Field Definitions**

| Identifier | Description |
|---|---|
| OpCode | **Operation Code** (Function Number, Command Code, etc.)<br>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**<br>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 G-5. UNDI CDB Field Definitions** (continued)

| Identifier | Description |
|---|---|
| CPBsize | **Command Parameter Block Size**<br>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.<br>The contents of the CPB structure will not be modified. |
| DBsize | **Data Block Size**<br>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**<br>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**<br>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**<br>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**<br>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.<br>Bits 0x0F & 0x0E: Command state: Not started (0), Queued (1), Error (2), Complete (3). |

continued

**Table G-5. UNDI CDB Field Definitions** (continued)

| Identifier | Description |
|---|---|
| IFnum | **Interface Number**<br>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**<br>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`.<br>Bit 0x00: Another CDB follows this one (1) or this is the last or only CDB in the list (0).<br>Bit 0x01: Queue command if busy (1), fail if busy (0). |

## G.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.

### G.3.1 Portability Macros

These macros are used for storage and communication portability.

#### G.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
```

#### G.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
```

### G.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')
```

### G.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
```

### G.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
```

### G.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
```

## G.3.2 Miscellaneous Macros

### G.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
```

## G.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.

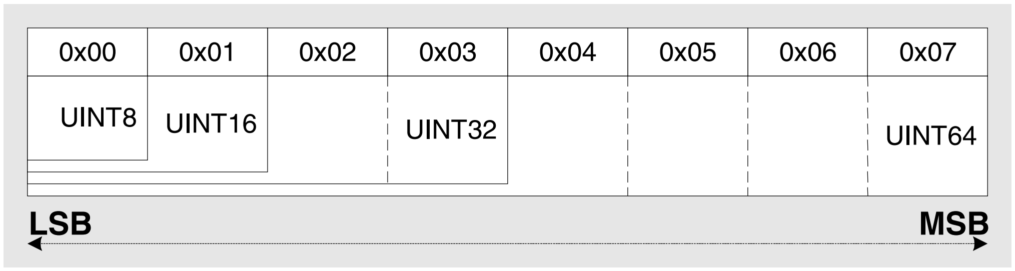

**Figure G-5. Storage Types**

### G.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
```

### G.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
```

### G.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;
```

### G.3.3.4 PXE_UINT8

Unsigned 8-bit integer.

```
typedef unsigned char PXE_UINT8;
```

### G.3.3.5 PXE_UINT16

Unsigned 16-bit integer.

```
typedef unsigned short PXE_UINT16;
```

### G.3.3.6 PXE_UINT32

Unsigned 32-bit integer.

```
typedef unsigned PXE_UINT32;
```

### G.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
```

### G.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;
```

### G.3.4 Simple Types

The PXE simple types are defined using one of the portability types from the previous section.

#### G.3.4.1 PXE_BOOL

Boolean (true/false) data type. For PXE zero is always false and non-zero is always true.

```
typedef PXE_UINT8 PXE_BOOL;
#define PXE_FALSE    0  // zero
#define PXE_TRUE     (!PXE_FALSE)
```

#### G.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

// Re-initialize 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 non-volatile 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
```

### G.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 non-volatile 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
```

### G.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.
```

### G.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
```

### G.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
```

### G.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
```

### G.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
```

### G.3.4.9 PXE_IPV4

This storage type is always big endian (network order) not little endian (Intel order).

```
typedef PXE_UINT32 PXE_IPV4;
```

### G.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;
```

### G.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;
```

### G.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
```

### G.3.5 Compound Types

All PXE structures must be byte packed.

#### G.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
```

### G.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()
```

### G.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                                    0x00

// 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
```

### G.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()
```

### G.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()
```

### G.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()
```

## G.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.

The figure below describes the different UNDI states (Stopped, Started and Initialized), shows the transitions between the states and which UNDI commands are valid in each state.

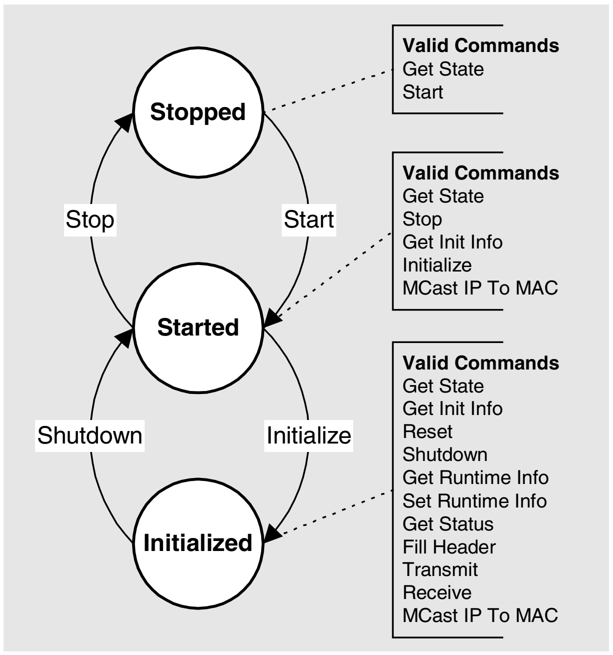

**Figure G-6. UNDI States, Transitions & Valid Commands**

**Note:** Additional requirements for S/W UNDI implementations: CPU register contents must be unchanged by S/W UNDI command execution (the application/driver does not have to save CPU registers when calling S/W UNDI). CPU arithmetic flags are undefined (application/driver must save CPU arithmetic flags if needed). Application/driver must remove CDB address from stack after control returns from S/W UNDI.

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.

## G.4.1 Command Linking & 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. The Link bit must be set in all other CDBs in the list.

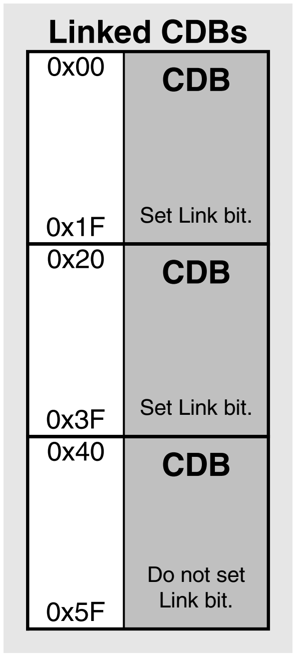

**Figure G-7. Linked CDBs**

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.

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.

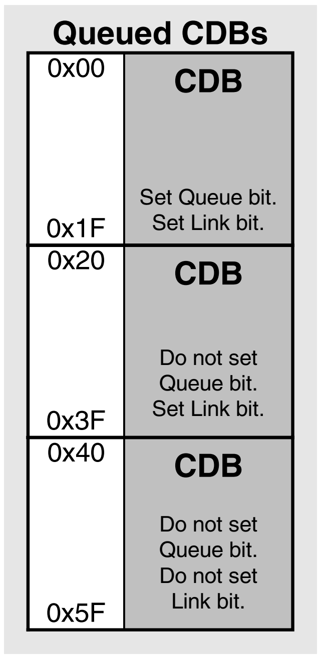

**Figure G-8. Queued CDBs**

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.

## G.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.

### G.4.2.1 Issuing the Command

To issue a Get State 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 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. |

### G.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. |

### G.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. |

## G.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. If this is a S/W UNDI, the Delay() and Virt2Phys() functions will not be called by this command.

### G.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 multi-threaded/multi-processor 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 non-zero 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_UINT64 Mem_IO;
  } PXE_CPB_START;
  #pragma pack()

  #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
```

#### G.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. |

#### G.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. |

### G.4.4 Stop

This command is used to change the UNDI operational state from started to stopped.

### G.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. |

### G.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. |

### G.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. |

### G.4.5 Get Init Info

This command is used to retrieve initialization information that is needed by drivers and applications to initialized UNDI.

### G.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. |

### G.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. |

### G.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 non-volatile storage items.

  PXE_UINT32 NvCount;

  // Width of non-volatile 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 non-zero,
  // 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.  See PXE_DUPLEX_xxxxx #defines below.

  PXE_UINT8 Duplex;

  // Supported loopback options.  See PXE_LOOPBACK_xxxxx #defines
  // below.

  PXE_UINT8 LoopBack;
} PXE_DB_GET_INIT_INFO;
#pragma pack()
```

```
#define PXE_MAX_TXRX_UNIT_ETHER                   1500
#define PXE_HWADDR_LEN_ETHER                      0x0006

#define PXE_DUPLEX_ENABLE_FULL_SUPPORTED          1
#define PXE_DUPLEX_FORCE_FULL_SUPPORTED           2
#define PXE_LOOPBACK_INTERNAL_SUPPORTED           1
#define PXE_LOOPBACK_EXTERNAL_SUPPORTED           2
```

## G.4.6 Get Config Info

This command is used to retrieve configuration information about the NIC being controlled by the UNDI.

### G.4.6.1 Issuing the Command

To issue a Get Config 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 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. |

### G.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. |

### G.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()
```

### G.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 (non-swappable) memory may need to be allocated. 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.

#### G.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*

Ccable 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 non-zero, 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.

 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 non-zero, 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.

    PXE_UINT8 Duplex;

    PXE_UINT8 LoopBack;
  } PXE_CPB_INITIALIZE;
  #pragma pack()

  #define PXE_DUPLEX_DEFAULT          0x00
  #define PXE_FORCE_FULL_DUPLEX       0x01
  #define PXE_ENABLE_FULL_DUPLEX      0x02

  #define LOOPBACK_NORMAL        0
  #define LOOPBACK_INTERNAL      1
  #define LOOPBACK_EXTERNAL      2
```

### G.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. |

### G.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 non-volatile 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()
```

## G.4.8 Reset

This command resets the network adapter and re-initializes 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.

### G.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`

### G.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. |

### G.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 non-volatile 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`

### G.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 re-assigned.

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.

### G.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. |

### G.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. |

### G.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. |

## G.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.

### G.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.

### G.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. |

### G.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!

## G.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.

### G.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 mutlicast 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;
```

## G.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. |

## G.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;
```

## G.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.

### G.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;
```

### G.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. |

### G.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;
```

## G.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.

### G.4.13.1 Issuing the Command

To issue a Statistics command, create a CDB and fill it in as shows 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**

### G.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. |

### G.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
```

### G.4.14 MCast IP To MAC

Translate a multicast IPv4 or IPv6 address to a multicast MAC address.

#### G.4.14.1 Issuing the Command

To issue a MCast IP To MAC command, create a CDB and fill it in as shows 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;
```

### G.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. |

### G.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;
```

## G.4.15 NvData

This command is used to read and write (if supported by NIC H/W) non-volatile storage on the NIC. Non-volatile storage could be EEPROM, FLASH or battery backed RAM.

### G.4.15.1 Issuing the Command

To issue a NvData command, create a CDB and fill it in as shows 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 non-volatile data CPBs, one for sparse updates and one for bulk updates. Sparse updates allow updating of single non-volatile storage items. Bulk updates always update all non-volatile storage items. Check the `!PXE.Implementation` flags to see which type of non-volatile update is supported by this UNDI and network device.

If you do not need to update the non-volatile 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 {

        // Non-volatile 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 non-volatile 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;
```

## G.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. Non-volatile 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. |

## G.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. Non-volatile 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 non-volatile storage items. This information is returned by the Get Init Info command.

```
typedef struct s_pxe_db_nvdata {

 // Arrays of data items from non-volatile 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;
```

## G.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.

### G.4.16.1 Issuing the Command

To issue a Get Status 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 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`

### G.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. |

### G.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 TxBuffer[n];
} PXE_DB_GET_STATUS;
#pragma pack()
```

## G.4.17 Fill Header

This command is used to fill the media header(s) in transmit packet(s).

## G.4.17.1 Issuing the Command

To issue a Fill Header command, create a CDB and fill it in as shows 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.

#### Non-Fragmented 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()
```

## G.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. |

### G.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. |

## G.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.

### G.4.18.1 Issuing the Command

To issue a Transmit command, create a CDB and fill it in as shows 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.

#### Non-Fragmented 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.
  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.
       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()
```

### G.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 re-used. |
| COMMAND_FAILED | Command failed. StatCode field contains error code. |
| COMMAND_QUEUED | Command has been queued. |
| INITIALIZE | Command has been not executed or queued. |

### G.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 re-used. |
| 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. |

## G.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.

### G.4.19.1 Issuing the Command

To issue a Receive 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 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.

```
#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.

  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()
```

## G.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. |

## G.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()
```

## G.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 *undi*config_table {

UINT32 NumberOfInterfaces; // the number of NIC devices that this UNDI controls

UINT32 reserved;

struct *undiconfig*table *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 though 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 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 address when the event occurs for the universal protocol driver to be able to use UNDI.

# H Index