



# **Extensible Firmware Interface Specification**

***Draft for Review***

Version 0.92

January 19, 2000

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

A license is hereby granted to copy and reproduce this specification for internal use only.

No other license, express or implied, by estoppel or otherwise, to any other intellectual property rights is granted herein.

Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information in this specification. Intel does not warrant or represent that such implementation(s) will not infringe such rights.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

This document contains information on products in the design phase of development. Do not finalize a design with this information. Revised information will be published when the product is available. Verify with your local sales office that you have the latest datasheet or specification before finalizing a design.

† Third-party trademarks are the property of their respective owners.

Intel order number 731843-001

Copyright © 1998, 1999, 2000. Intel Corporation, All Rights Reserved.

## Revision History

Revision	Revision History	Date	Authors
0.92	Revised Chapter 14 (PXE Base Code Protocol) and Chapter 15 (Simple Network Protocol). At the request of Compaq, all references to Alpha processors were deleted. Deleted <b>LocateProtocol()</b> boot services function. Added terms to Glossary. Made major language edits in Chapter 3 and minor language edits in other chapters. Updated Unicode Collation Protocol for FAT32 file system support.	1/19/00	Andrew J. Fish, Mark Doran, Ken Reneris, Mani Ayyar, Michael Kinney, Gary Taylor
0.91	Added new boot services functions (WaitforEvent, UnloadImage) and new runtime services functions (SetVirtualAddressMap, ConvertPointer). Modified SIMPLE_INPUT_INTERFACE protocol. Added SIMPLE_TEXT_OUTPUT_MODE structure to SIMPLE_TEXT_OUTPUT protocol. Added EFI_BLOCK_IO_MEDIA structure to BLOCK_IO protocol. Added SERIAL_IO_MODE structure to SERIAL_IO protocol. Added EFI_PXE_MEDIA structure to PXE_IO protocol. Made minor changes in File System Format. Added information on Alpha processor support. Changed EFI_MEMORY_DESCRIPTOR structure. Updated EFI Partitioning structures.	7/30/99	Andrew J. Fish, Mark Doran, Ken Reneris, Mani Ayyar, Michael Kinney, Gary Taylor
0.90	Renamed "Extensible Firmware Interface Specification" (formerly "Intel Boot Initiative Specification"). Reorganized extensively. On disk format changes. Serial protocol added. Expanded description of boot selection management.	4/30/99	Andrew J. Fish, Mark Doran, Ken Reneris, Mani Ayyar, Michael Kinney, Gary Taylor
0.70	Relocatable image format included. New boot options interfaces and general purpose NVRAM interfaces added. Network boot protocols for PXE and device I/O protocol added. LoadImage interface added and changes made to ExitEFIImage (now Exit( )) interface. Changed memory allocation and added pool interfaces. Added system reset API. Removed legacy BIOS API information. Augmented Device Path. Added Appendix for EFI console definition.	12/21/98	Andrew J. Fish, Mark Doran, Ken Reneris, Mani Ayyar
0.35	File system and I/O protocols added. Removable media boot strategy included. Some simplification made to file system structures. Added device path definition (Appendix D). Revised goals section. Added glossary. Significant format clean-up.	10/23/98	Andrew J. Fish, Mark Doran, Ken Reneris, Mani Ayyar
0.31	Editing corrections and enhancements (no technical changes).	9/29/98	Andrew J. Fish, Mark Doran, N. Navarro
0.30	Initial review draft	9/25/98	Andrew J. Fish, Mark Doran



# Table of Contents

---

## 1 Introduction

1.1	Overview.....	2
1.2	Goals .....	3
1.3	Target Audience .....	5
1.4	Related Information .....	5
1.5	Prerequisite Specifications .....	7
1.5.1	ACPI Specification .....	7
1.5.2	WfM Specification .....	7
1.5.3	Additional Considerations for IA-64 Platforms .....	8
1.6	EFI Design Overview .....	8
1.7	Migration Requirements.....	10
1.7.1	Legacy Operating System Support.....	10
1.7.2	Supporting the EFI Specification on a Legacy Platform.....	10
1.8	Conventions Used in This Document.....	11
1.8.1	Data Structure Descriptions.....	11
1.8.2	Typographic Conventions .....	11
1.9	Guidelines for Use of the Term “Extensible Firmware Interface” .....	11

## 2 Overview

2.1	Boot Manager .....	14
2.2	Firmware Core .....	14
2.2.1	EFI Services .....	14
2.2.2	Runtime Services.....	15
2.3	Calling Conventions.....	15
2.3.1	Data Types .....	16
2.3.2	IA-32 Platforms.....	17
2.1.3	IA-64 Platforms.....	18
2.4	Protocols.....	18

2.5	Requirements .....	20
2.5.1	Required Elements .....	21
2.5.2	Optional Elements .....	21
2.5.3	Appendixes .....	22

### 3 Services

3.1	Event, Timer, and Task Priority Services .....	24
3.1.1	CreateEvent() .....	26
3.1.2	CloseEvent() .....	30
3.1.3	SignalEvent() .....	31
3.1.4	WaitForEvent() .....	32
3.1.5	SetTimer() .....	33
3.1.6	RaiseTPL() .....	35
3.1.7	RestoreTPL() .....	37
3.2	Memory Allocation Services .....	38
3.2.1	AllocatePages() .....	41
3.2.2	FreePages() .....	44
3.2.3	GetMemoryMap() .....	45
3.2.4	AllocatePool() .....	49
3.2.5	FreePool() .....	50
3.3	Protocol Handler Services .....	51
3.3.1	InstallProtocolInterface() .....	53
3.3.2	UninstallProtocolInterface() .....	55
3.3.3	ReinstallProtocolInterface() .....	56
3.3.4	RegisterProtocolNotify() .....	57
3.3.5	LocateHandle() .....	58
3.3.6	HandleProtocol() .....	60
3.3.7	LocateDevicePath() .....	61
3.4	Image Services .....	63
3.4.1	LoadImage() .....	65
3.4.2	StartImage() .....	67
3.4.3	UnloadImage() .....	68
3.4.4	EFI_IMAGE_ENTRY_POINT .....	69

3.4.5	Exit() .....	70
3.4.6	ExitBootServices() .....	72
3.5	Variable Services .....	73
3.5.1	GetVariable() .....	74
3.5.2	GetNextVariableName() .....	76
3.5.3	SetVariable() .....	78
3.6	Time Services .....	80
3.6.1	GetTime() .....	81
3.6.2	SetTime() .....	84
3.6.3	GetWakeupTime() .....	85
3.6.4	SetWakeupTime() .....	86
3.7	Virtual Memory Services .....	87
3.7.1	SetVirtualAddressMap() .....	88
3.7.2	ConvertPointer() .....	90
3.8	Miscellaneous Services .....	91
3.8.1	ResetSystem() .....	92
3.8.2	SetWatchdogTimer() .....	94
3.8.3	Stall() .....	96
3.8.4	GetNextMonotonicCount() .....	97
3.8.5	GetNextHighMonotonicCount() .....	98

## 4 EFI Image

4.1	LOADED_IMAGE Protocol .....	99
4.1.1	LOADED_IMAGE.Unload() .....	102
4.2	EFI Image Header .....	103
4.3	EFI Applications .....	104
4.4	EFI OS Loaders .....	104
4.5	EFI Drivers .....	104
4.5.1	EFI Image Handoff State .....	105
4.5.1.1	IA-32 Handoff State .....	109
4.5.1.2	IA-64 Handoff State .....	110

## 5 Device Path Protocol

5.1	Device Path Overview .....	111
5.2	EFI_DEVICE_PATH Protocol .....	112

5.3	Device Path Nodes .....	113
5.3.1	Generic Device Path Structures .....	113
5.3.2	Hardware Device Path .....	115
5.3.2.1	PCI Device Path .....	115
5.3.2.2	PCCARD Device Path .....	116
5.3.2.3	Memory Mapped Device Path .....	116
5.3.2.4	Vendor Device Path .....	116
5.3.3	ACPI Device Path .....	117
5.3.4	Messaging Device Path .....	117
5.3.4.1	ATAPI Device Path .....	118
5.3.4.2	SCSI Device Path .....	118
5.3.4.3	Fibre Channel Device Path .....	118
5.3.4.4	1394 Device Path .....	119
5.3.4.5	USB Device Path .....	119
5.3.4.6	I <sub>2</sub> O Device Path .....	119
5.3.4.7	MAC Address Device Path .....	120
5.3.4.8	IPv4 Device Path .....	120
5.3.4.9	IPv6 Device Path .....	121
5.3.4.10	InfiniBand <sup>†</sup> Device Path .....	121
5.3.4.11	UART Device Path .....	122
5.3.4.12	Vendor-Defined Messaging Device Path .....	123
5.3.5	Media Device Path .....	123
5.3.5.1	Hard Drive .....	123
5.3.5.2	CD-ROM Media Device Path .....	125
5.3.5.3	Vendor-Defined Media Device Path .....	125
5.3.5.4	File Path Media Device Path .....	126
5.3.5.5	Media Protocol .....	126
5.3.6	BIOS Boot Specification Device Path .....	126
5.4	Device Path Generation Rules .....	127
5.4.1	Housekeeping Rules .....	127
5.4.2	Rules with ACPI _HID and _UID .....	128
5.4.3	Rules with ACPI _ADR .....	128
5.4.4	Hardware vs. Messaging Device Path Rules .....	129



5.4.5	Media Device Path Rules .....	129
5.4.6	Other Rules .....	130
<b>6 Device I/O Protocol</b>		
6.1	Device I/O Overview .....	131
6.2	DEVICE_IO Protocol .....	132
6.2.1	DEVICE_IO.Mem(), .Io(), and .Pci().....	135
6.2.2	DEVICE_IO.PciDevicePath() .....	137
6.2.3	DEVICE_IO.Map() .....	138
6.2.4	DEVICE_IO.Unmap().....	140
6.2.5	DEVICE_IO.AllocateBuffer() .....	141
6.2.6	DEVICE_IO.Flush() .....	143
<b>7 Console I/O Protocol</b>		
7.1	Console I/O Overview .....	145
7.2	ConsoleIn Definition.....	146
7.3	SIMPLE_INPUT Protocol.....	148
7.3.1	SIMPLE_INPUT.Reset() .....	149
7.3.2	SIMPLE_INPUT.ReadKeyStroke() .....	150
7.4	ConsoleOut or StandardError .....	151
7.5	SIMPLE_TEXT_OUTPUT Protocol.....	151
7.5.1	SIMPLE_TEXT_OUTPUT.Reset() .....	154
7.5.2	SIMPLE_TEXT_OUTPUT.OutputString() .....	155
7.5.3	SIMPLE_TEXT_OUTPUT.TestString() .....	157
7.5.4	SIMPLE_TEXT_OUTPUT.QueryMode().....	158
7.5.5	SIMPLE_TEXT_OUTPUT.SetMode() .....	159
7.5.6	SIMPLE_TEXT_OUTPUT.SetAttribute().....	160
7.5.7	SIMPLE_TEXT_OUTPUT.ClearScreen() .....	162
7.5.8	SIMPLE_TEXT_OUTPUT.SetCursorPosition().....	163
7.5.9	SIMPLE_TEXT_OUTPUT.EnableCursor() .....	164
<b>8 Block I/O Protocol</b>		
8.1	BLOCK_IO Protocol.....	165
8.1.1	EFI_BLOCK_IO.Reset() .....	168
8.1.2	EFI_BLOCK_IO.ReadBlocks().....	169

8.1.3	EFI_BLOCK_IO.WriteBlocks()	171
8.1.4	BLOCK_IO.FlushBlocks()	173

## 9 Disk I/O Protocol

9.1	DISK_IO Protocol	175
9.1.1	EFI_DISK_IO.ReadDisk()	177
9.1.2	EFI_DISK_IO.WriteDisk()	178

## 10 File System Protocol

10.1	SIMPLE_FILE_SYSTEM Protocol	179
10.1.1	EFI_FILE_IO.OpenVolume()	181
10.2	EFI_FILE_HANDLE Protocol	182
10.2.1	EFI_FILE.Open()	184
10.2.2	EFI_FILE.Close()	187
10.2.3	EFI_FILE.Delete()	188
10.2.4	EFI_FILE.Read()	189
10.2.5	EFI_FILE.Write()	190
10.2.6	EFI_FILE.SetPosition()	191
10.2.7	EFI_FILE.GetPosition()	192
10.2.8	EFI_FILE.GetInfo()	193
10.2.9	EFI_FILE.SetInfo()	194
10.2.10	EFI_FILE.Flush()	195
10.2.11	EFI_GENERIC_FILE_INFO	196
10.2.12	EFI_FILE_SYSTEM_INFO	198

## 11 Load File Protocol

11.1	LOAD_FILE Protocol	199
11.1.1	LOAD_FILE.LoadFile()	200

## 12 SERIAL I/O Protocol

12.1	SERIAL_IO Protocol	203
12.1.1	SERIAL_IO.Reset()	207
12.1.2	SERIAL_IO.SetAttributes()	208
12.1.3	SERIAL_IO.SetControl()	210
12.1.4	SERIAL_IO.GetControl()	211
12.1.5	SERIAL_IO.Write()	212
12.1.6	SERIAL_IO.Read()	213

## 13 Unicode Collation Protocol

13.1	UNICODE_COLLATION Protocol.....	215
13.1.1	UNICODE_COLLATION.StriColl() .....	217
13.1.2	UNICODE_COLLATION.MetaiMatch() .....	218
13.1.3	UNICODE_COLLATION.StrLwr() .....	220
13.1.4	UNICODE_COLLATION.StrUpr() .....	221
13.1.5	UNICODE_COLLATION.FatToStr().....	222
13.1.6	UNICODE_COLLATION.StrToFat().....	223

## 14 PXE Base Code Protocol

14.1	EFI_PXE_BASE_CODE Protocol.....	225
14.1.1	EFI_PXE_BASE_CODE.Start() .....	232
14.1.2	EFI_PXE_BASE_CODE.Stop() .....	234
14.1.3	EFI_PXE_BASE_CODE.Dhcp() .....	235
14.1.4	EFI_PXE_BASE_CODE.Discover().....	237
14.1.5	EFI_PXE_BASE_CODE.Mtftp().....	240
14.1.6	EFI_PXE_BASE_CODE.UdpWrite().....	243
14.1.7	EFI_PXE_BASE_CODE.UdpRead().....	245
14.1.8	EFI_PXE_BASE_CODE.SetIpFilter() .....	248
14.1.9	EFI_PXE_BASE_CODE.Arp() .....	249
14.1.10	EFI_PXE_BASE_CODE.SetParameters().....	250
14.1.11	EFI_PXE_BASE_CODE.SetStationIp() .....	251
14.1.12	EFI_PXE_BASE_CODE.SetPackets().....	252
14.1.13	EFI_PXE_BASE_CODE_CALLBACK .....	253

## 15 SIMPLE\_NETWORK Protocol

15.1	EFI_SIMPLE_NETWORK Protocol.....	255
15.1.1	EFI_SIMPLE_NETWORK.Start().....	260
15.1.2	EFI_SIMPLE_NETWORK.Stop().....	261
15.1.3	EFI_SIMPLE_NETWORK.Initialize() .....	262
15.1.4	EFI_SIMPLE_NETWORK.Reset() .....	263
15.1.5	EFI_SIMPLE_NETWORK.Shutdown() .....	264
15.1.6	EFI_SIMPLE_NETWORK.ReceiveFilters().....	265
15.1.7	EFI_SIMPLE_NETWORK.StationAddress() .....	267
15.1.8	EFI_SIMPLE_NETWORK.Statistics() .....	268

15.1.9	EFI_SIMPLE_NETWORK.MCastIPtoMAC()	270
15.1.10	EFI_SIMPLE_NETWORK.NVData()	271
15.1.11	EFI_SIMPLE_NETWORK.GetStatus()	273
15.1.12	EFI_SIMPLE_NETWORK.Transmit()	275
15.1.13	EFI_SIMPLE_NETWORK.Receive()	277

## 16 File System Format

16.1	System Partition	279
16.1.1	File System Format	280
16.1.2	File Names	280
16.1.3	Directory Structure	280
16.2	Partition Discovery	282
16.2.1	Extensible Firmware Interface Partition Header	283
16.2.2	ISO-9660 and El Torito	287
16.2.3	Legacy Master Boot Record	288
16.2.4	Legacy Master Boot Record and EFI Partitions	289
16.3	Media Formats	290
16.3.1	Removable Media	290
16.3.2	Diskette	291
16.3.3	Hard Drive	291
16.3.4	CD-ROM and DVD-ROM	291

## 17 Boot Manager

17.1	Firmware Boot Manager	293
17.2	Globally-Defined Variables	296
17.3	Boot Mechanisms	298
17.3.1	Boot via SIMPLE_FILE_PROTOCOL	298
17.3.2	Boot via LOAD_FILE Protocol	298
17.3.2.1	Network Booting	298
17.3.2.2	Future Boot Media	298

## 18 PCI Expansion ROM

18.1	Standard PCI Expansion ROM Header	299
18.2	EFI PCI Expansion ROM Header	300
18.3	Multiple Image Format Support	301
18.4	EFI PCI Expansion ROM Driver	301

<b>A GUID and Time Formats .....</b>	<b>203</b>
<b>B Console</b>	
B.1    SIMPLE_INPUT .....	305
B.2    SIMPLE_TEXT_OUTPUT .....	306
<b>C Device Path Examples</b>	
C.1    Example Computer System .....	309
C.2    Legacy Floppy .....	310
C.3    IDE Disk.....	311
C.4    Secondary Root PCI Bus with PCI to PCI Bridge .....	312
C.5    ACPI Terms .....	314
C.6    EFI Device Path as a Name Space .....	315
<b>D Status Codes .....</b>	<b>317</b>
<b>E Alphabetic Function List .....</b>	<b>319</b>
<b>F Glossary .....</b>	<b>327</b>
<b>G Index.....</b>	<b>335</b>

## Figures

1-1.	EFI Conceptual Overview .....	9
2-1.	Boot Sequence .....	13
2-1.	Construction of a Protocol .....	19
3-1.	Device Handle to Protocol Handler Mapping .....	52
4-1.	Stack after ImageEntryPoint Called, IA-32 .....	109
4-2.	Stack after ImageEntryPoint Called, IA-64 .....	110
16-1.	Nesting of Legacy MBR Partition Records.....	282
16-2.	EFI Partitioning Scheme.....	284
C-1.	Example Computer System .....	309
C-2.	Partial ACPI Name Space for Example System.....	310
C-3.	EFI Device Path Displayed As a Name Space .....	315

## Tables

1-1.	Organization of EFI Specification.....	2
2-1.	EFI Runtime Services .....	15
2-2.	Common EFI Data Types .....	16
2-3.	Modifiers for Common EFI Data Types.....	17
2-4.	EFI Protocols .....	20
2-5.	Required EFI Implementation Elements .....	21
2-6.	Optional EFI Implementation Elements .....	22
3-1.	Event, Timer, and Task Priority Functions .....	24
3-2.	TPL Restrictions .....	25
3-3.	Memory Allocation Functions.....	38
3-4.	Memory Type Usage Before ExitBootServices().....	39
3-5.	Memory Type Usage After ExitBootServices().....	40
3-6.	Protocol Interface Functions .....	51
3-7.	Image Type Differences Summary .....	63
3-8.	Image Functions .....	64
3-9.	Variable Services Functions .....	73
3-10.	Time Services Functions .....	80
3-11.	Virtual Memory Functions .....	87
3-12.	Miscellaneous Services Functions.....	91

5-1.	Generic Device Path Node Structure .....	114
5-2.	Device Path End Structure .....	115
5-3.	PCI Device Path .....	115
5-4.	PCCARD Device Path .....	116
5-5.	Memory Mapped Device Path.....	116
5-6.	Vendor-Defined Device Path .....	116
5-7.	ACPI Device Path .....	117
5-8.	ATAPI Device Path .....	118
5-9.	SCSI Device Path .....	118
5-10.	Fibre Channel Device Path.....	118
5-11.	1394 Device Path .....	119
5-12.	USB Device Path .....	119
5-13.	I <sub>2</sub> O Device Path.....	119
5-14.	MAC Address Device Path .....	120
5-15.	IPv4 Device Path .....	120
5-16.	IPv6 Device Path .....	121
5-17.	InfiniBand Device Path .....	121
5-18.	UART Device Path.....	122
5-19.	Vendor-Defined Messaging Device Path .....	123
5-20.	Hard Drive Media Device Path.....	124
5-21.	CD-ROM Media Device Path .....	125
5-22.	Vendor-Defined Media Device Path.....	125
5-23.	File Path Media Device Path .....	126
5-24.	Media Protocol Media Device Path.....	126
5-25.	BIOS Boot Specification Device Path .....	127
5-26.	ACPI _CRS to EFI Device Path Mapping .....	128
5-27.	ACPI _ADR to EFI Device Path Mapping .....	129
6-1.	PCI Address.....	136
7-1.	Supported Unicode Control Characters .....	146
7-2.	EFI Scan Codes for SIMPLE_INPUT_INTERFACE .....	146
16-1.	EFI Partition Header .....	285
16-2.	EFI Partition Entry.....	286
16-3.	Defined EFI Partition Entry Type GUID .....	287
16-4.	Defined EFI Partition Entry Attributes .....	287

16-5.	Legacy Master Boot Record .....	288
16-6.	Legacy Master Boot Record Partition Record.....	289
16-7.	Required Legacy Master Boot Record Entry to Precede an EFI Partition Header .....	290
17-1	Global Variables .....	296
18-1.	Standard PCI Expansion ROM Header .....	299
18-2.	PCI Data Structure.....	300
18-3.	EFI PCI Expansion ROM Header .....	301
A-1.	EFI GUID Format.....	203
B-1.	EFI Scan Codes for SIMPLE_INPUT.....	305
B-2.	Control Sequences that Can Be Used to Implement SIMPLE_TEXT_OUTPUT .....	306
C-1.	Legacy Floppy Device Path .....	311
C-2.	IDE Disk Device Path .....	312
C-3.	Secondary Root PCI Bus with PCI to PCI Bridge Device Path .....	313
D-1.	EFI_STATUS Success Codes .....	317
D-2.	EFI_STATUS Error Codes.....	317
D-3.	EFI_STATUS Warning Codes .....	318
E-1.	Alphabetic Function List.....	319





# 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 IA-64 Intel Architecture-based computers. Thus, the specification is applicable to a full range of hardware platforms from mobile systems to servers. 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_BC Protocol	Defines the PXE_BC protocol, which is used perform network boot operations.

continued

**Table 1-1. Specification Organization and Contents** (continued)

Chapter/Appendix	Description
15. Simple Network Protocol	Defines the Simple Network Protocol, which provides a packet level interface to a network device.
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. 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, 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<sup>†</sup> Windows NT<sup>†</sup> Server*, Intel Corporation, Microsoft Corporation, 1998, <http://developer.intel.com/design/servers/desguide/index.htm>
- *IA-64 Software Conventions and Runtime Architecture Guide, Rev. 2.5E (SC-2847)*. Also available at <http://developer.intel.com/design/ia64/downloads/245256.htm>
- *IA-64 System Abstraction Layer Specification*, Rev 2.6, document number SC-2793, 1997, 1998 Intel Corporation
- *IA-64 Processor Programmer’s Reference Manual*, Rev 4.0. Document number SC-2766-9. Intel Corporation 1997, 1998
- *IA-64 Application Developer’s Architecture Guide*. Intel order number 245188-001. Also available at <http://developer.intel.com/design/ia64/downloads/adag.htm>
- *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*, 10/96
- *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/specs.html>
- *Portable Executable and Common Object File Format Specification*. See [http://msdn.microsoft.com/library/specs/msdn\\_pecoff.htm](http://msdn.microsoft.com/library/specs/msdn_pecoff.htm)
- *Preboot Execution Environment (PXE) Specification*, Version 1.0 Release Candidate. Intel Corporation, 1998
- *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) Specification*, <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) Specification* 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 IA-64 Platforms

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

The following specifications are the required IA-64 architectural specifications for all IA-64 platforms:

- *IA-64 System Abstraction Layer Specification*, Rev 2.6, (document number SC-2328)
- *Merced Programmer's Reference Manual*, Volume 1, Rev 2.0 (document number SC-2084-6)

## 1.6 EFI Design Overview

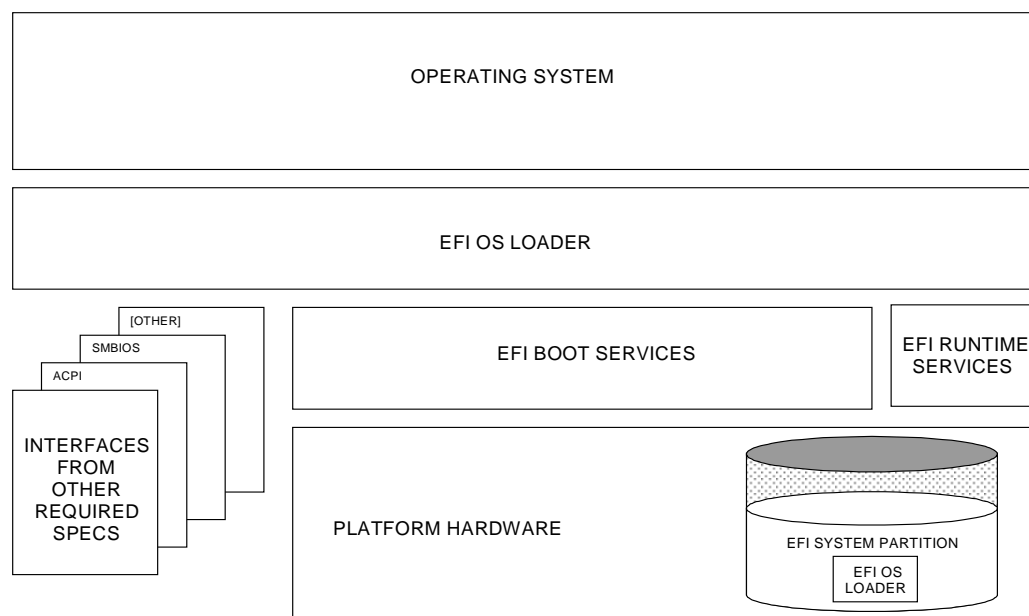
The design of EFI is based on the following fundamental elements:

- *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 EFI 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. Processors of the IA-64 family 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:

<b>Prototype</b>	This typeface is use to indicate prototype code.
<i>Argument</i>	This typeface is used to indicate arguments.
<b>Name</b>	This typeface is used to indicate actual code or a programming construct.
<b>register</b>	This typeface is used to indicate a processor register.

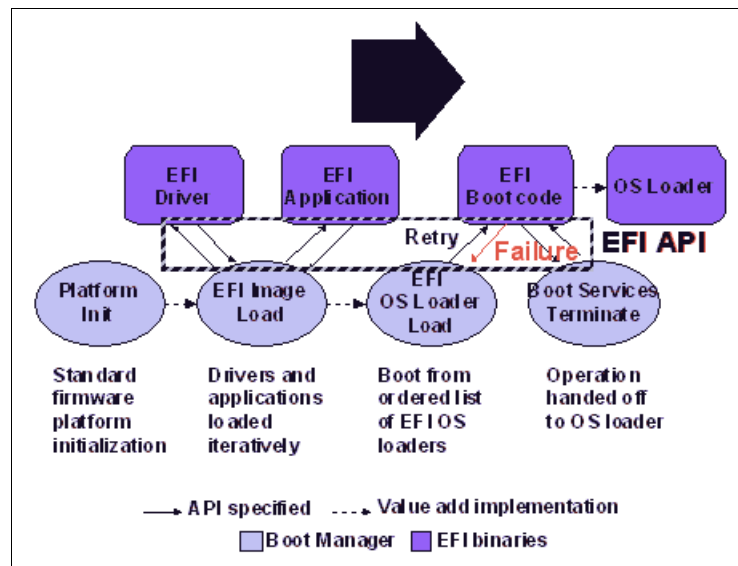
## 1.9 Guidelines for Use of the Term “Extensible Firmware Interface”

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.



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. 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 64-bit 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 a 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 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.

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. IA-64 and IA-32 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 <b>FALSE</b> or a 1 for <b>TRUE</b> . Other values are undefined.
INTN	Signed value of native width. (4 bytes on IA-32, 8 bytes on IA-64)
UINTN	Unsigned value of native width. (4 bytes on IA-32, 8 bytes on IA-64)
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.
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	Handle to a device driver. 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.
<Enumerated Type>	Signed value of native width. (4 bytes in AIA-32, 8 bytes on IA-64)



**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 <b>NULL</b> 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 IA-64 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 IA-64. Refer to the document *64 Bit Runtime Architecture and Software Conventions for IA-64* for more information.

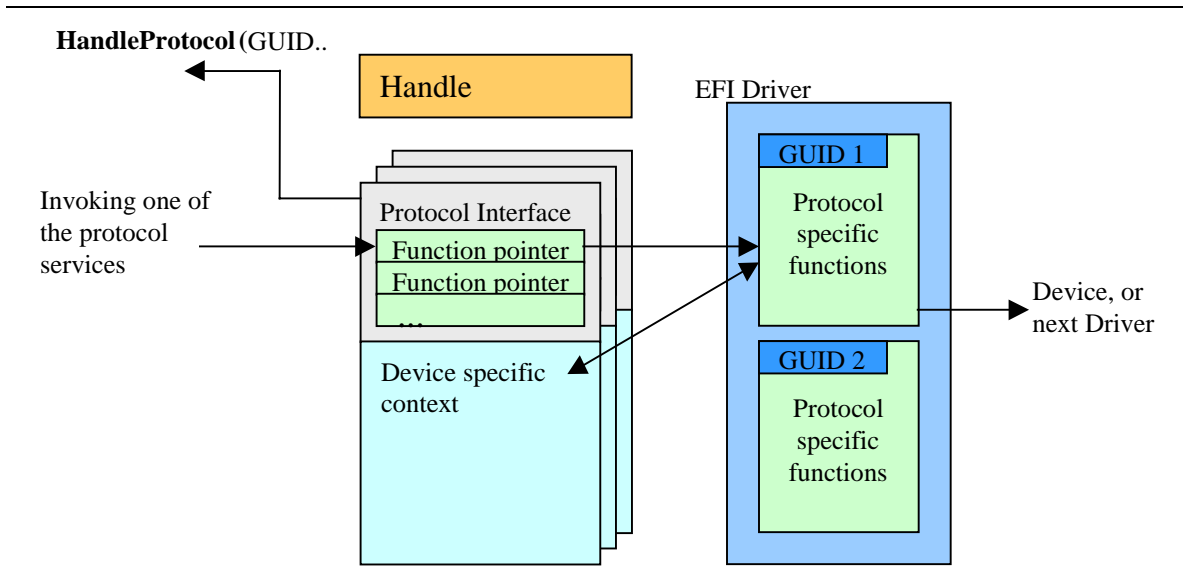
## 2.4 Protocols

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 device 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 2-1 shows the construction of a protocol. The EFI driver contains functions specific to one or more protocol implementations, and registers them with **InstallProtocol()** 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 2-1. 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,
    &IllustrationProtocol,
    &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.

**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	Functionality to provide BLOCK_IO interfaces for logical block devices as defined by partition table, or El Torito “no emulation” device.
BLOCK_IO protocol	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	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	Protocol interfaces for opening disk volumes through a DISK_IO interface.
EFI_FILE_HANDLE protocol	Protocol interfaces for accessing the device with file I/O style accesses through a DISK_IO interface.
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	Protocol interfaces for Unicode string comparison operations.

## 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.

### 2.5.3 Appendixes

The content of Appendixes B, C and D 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.

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* any 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 0)
- 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.
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:

- **TPL\_NORMAL**, the lowest level 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\_NORMAL** 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.

Most event notifications, including timers, occur at the **TPL\_NOTIFY** level and therefore interrupt normal execution. Normally executing code may temporarily raise the task priority level to the **TPL\_NOTIFY** level by using the **RaiseTPL()** function. By doing this, **TPL\_NOTIFY** level event notifications are prohibited from interrupting the code until that task priority level is restored and lowered below the **TPL\_NOTIFY** level by calling the **RestoreTPL()** function. Many of the EFI services and EFI protocols interface functions have restrictions on the TPL levels they may execute. See Table 3-2 for a summary of these restrictions.



**Table 3-2. TPL Restrictions**

Name	Restriction	TPL
Memory Allocation Services	<=	TPL_NOTIFY
Variable Services	<=	TPL_CALLBACK
ExitBootServices()	=	TPL_NORMAL
LoadImage()	<=	TPL_CALLBACK
WaitForEvent() Notifications	=	TPL_NORMAL
Event Notification Levels	<=	TPL_HIGH_LEVEL
Event Callout Notifications	>	TPL_NORMAL
Protocol Interface Functions	<=	TPL_NOTIFY
Block I/O Protocol	<=	TPL_CALLBACK
Disk I/O Protocol	<=	TPL_CALLBACK
Simple File System Protocol	<=	TPL_CALLBACK
Simple Input Protocol	<=	TPL_NORMAL
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

<i>Type</i>	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.
<i>NotifyTpl</i>	The task priority level of event notifications. See Section 0.
<i>NotifyFunction</i>	Pointer to the event’s notification function. See “Related Definitions”.
<i>NotifyContext</i>	Pointer to the notification function’s context; corresponds to parameter <i>Context</i> in the notification function.
<i>Event</i>	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_NOTIFY_PC_INTERFACE  0x00000400

#define EVT_SIGNAL_EXIT_BOOT_SERVICES  0x00000201
#define EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE  0x60000202

```

<b>EVT_TIMER</b>	The event is a timer event and may be passed to <b>SetTimer()</b> . Note that timers only function during boot services time.
<b>EVT_RUNTIME</b>	The event is allocated from runtime memory. If an event is to be signaled after the call to <b>ExitBootServices()</b> , the event's data structure and notification function need to be allocated from runtime memory. For more information, see <b>SetVirtualAddressMap()</b> (Section 0).
<b>EVT_RUNTIME_CONTEXT</b>	The event's <i>NotifyContext</i> pointer points to a runtime memory address. See the discussion of <b>EVT_RUNTIME</b> .
<b>EVT_NOTIFY_WAIT</b>	The event's <i>NotifyFunction</i> is to be invoked whenever the event is being waited on via <b>WaitForEvent()</b> .
<b>EVT_NOTIFY_SIGNAL</b>	The event's <i>NotifyFunction</i> is to be invoked whenever the event is signaled via <b>SignalEvent()</b> .
<b>EVT_NOTIFY_PC_INTERFACE</b>	The event's <i>NotifyFunction</i> is pcode.
<b>EVT_SIGNAL_EXIT_BOOT_SERVICES</b>	The event is to be notified by the system when <b>ExitBootServices()</b> is invoked. This type can not be used with any other EVT bit type.

**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 places a call to its notification function in a FIFO queue. There are three such queues, one for each of the task priority levels (*NotifyTpl*) defined in Section 0. The functions in these queues are invoked in FIFO order, starting with the highest priority level queue, when **RestoreTPL()** is called.

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 would link the event's notify function into the appropriate FIFO queue and call **RestoreTPL()**, as described above.

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**. As with asynchronous events, this changes each event's state to "signaled" and places a call to its notification function in the appropriate FIFO queue. Firmware then calls **RestoreTPL()**, which causes all the notification functions to be executed, and control eventually returns to the caller of **ExitBootServices()**.

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 0.

The **EVT\_NOTIFY\_WAIT** and **EVT\_NOTIFY\_SIGNAL** flags are exclusive. If neither flag is specified, it is assumed that the caller does not want 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()**). If the **EVT\_NOTIFY\_SIGNAL** flag is specified then the event's notify function is queued whenever the event is signaled.

When an event that does not have a notification function is signaled, the call to **WaitForEvent()** will return with an index indicating which event was signaled. Note that the event maintains its signaled state until a caller is notified.

Note: Since their internal structure is unknown to the caller, events cannot be modified by the caller. The only way to "change" an event is to close it and create another event that has the desired behavior.

## 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 handle is no longer valid and may not be used on any subsequent function calls.

#### Status Codes Returned

EFI_SUCCESS	The event has been closed.
EFI_INVALID_PARAMETER	<i>Event</i> was not a valid handle.

### 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 the event's notification function is scheduled to be invoked at the event's task priority level. **SignalEvent()** may be invoked from any task priority level.

#### Status Codes Returned

EFI_SUCCESS	The event was signaled.
EFI_INVALID_PARAMETER	<i>Event</i> was not a valid handle.

### 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 pointers to events. 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 that has a notification function is not permitted. If any event in *Event* has a notification function, **WaitForEvent()** returns **EFI\_INVALID\_PARAMETER** and sets *Index* to indicate which event caused the failure. The function must also be called at **TPL\_NORMAL**. If an attempt is made to call this functional any TPL other than **TPL\_NORMAL**, **EFI\_UNSUPPORTED** will be 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 posted to the list of events that are being checked.

#### Status Codes Returned

EFI_SUCCESS	The event indicated by <i>Index</i> was signaled.
EFI_INVALID_PARAMETER	The event indicated by <i>Index</i> has a notification function or <i>Event</i> was not a valid handle.
EFI_UNSUPPORTED	The current TPL is not <b>TPL_NORMAL</b> .



### 3.1.5 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

<i>Event</i>	The timer event that is to be signaled at the specified time. Type <b>EFI_EVENT</b> is defined in Section 3.1.1.
<i>Type</i>	The type of time that is specified in <i>TriggerTime</i> . See the timer delay types in “Related Definitions”.
<i>TriggerTime</i>	The number of 100ns units until the timer expires.

#### Related Definitions

```

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

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

### 3.1.6 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 the current task priority level. See “Related Definitions”.

#### Related Definitions

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

/** *****
// Task Priority Levels
// *****
#define TPL_NORMAL        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 the task's previous priority level.

Only three task priority levels are exposed outside of the firmware during EFI boot services execution. The first is **TPL\_NORMAL** 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 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: Applications may only use **TPL\_NORMAL**, **TPL\_CALLBACK**, and **TPL\_NOTIFY**. **TPL\_HIGH\_LEVEL** and all other values are reserved for use by the firmware; using them in an application will 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()**.

### 3.1.7 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 0.

#### Description

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

#### 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 3-3.

**Table 3-3. 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 3-4 lists each type and its usage before the call; Table 3-5 lists each type and its usage after the call.

**Table 3-4. Memory Type Usage Before ExitBootServices()**

<b>Mnemonic</b>	<b>Description</b>
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 Services 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 3-5. 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 <b>ExitBootServices()</b> is utilizing one or more <b>EfiLoaderCode</b> ranges.
EfiLoaderData	The Loader and/or OS may use this memory as they see fit. Note: the OS loader that called <b>ExitBootServices()</b> is utilizing one or more <b>EfiLoaderData</b> 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()** should first call **GetMemoryMap()** to obtain the current memory map. Following that 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

<i>Type</i>	The type of allocation to perform. See “Related Definitions”.
<i>MemoryType</i>	The type of memory to allocate. The only types allowed are <b>EfiLoaderData</b> , <b>EfiRuntimeServicesCode</b> , <b>EfiRuntimeServicesData</b> , <b>EfiBootServicesCode</b> , <b>EfiBootServicesData</b> . Normal allocations (that is, allocations by any EFI application) are of type <b>EfiLoaderData</b> . See “Related Definitions”, Table 3-4, and Table 3-5.
<i>Pages</i>	The number of contiguous 4KB pages to allocate.
<i>Memory</i>	Pointer to a physical address. On input, the way in which the address is used depends on the value of <i>Type</i> . 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 3-4 and Table 3-5.
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_INVALID_PARAMETER	<i>Memory</i> is not a page-aligned address or <i>Pages</i> 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

<i>MemoryMapSize</i>	A pointer to the size, in bytes, of the <i>MemoryMap</i> 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.
<i>MemoryMap</i>	A pointer to the buffer in which firmware places the current memory map. The map is an array of <b>EFI_MEMORY_DESCRIPTOR</b> s. See “Related Definitions”.
<i>MapKey</i>	A pointer to the location in which firmware returns the key for the current memory map.
<i>DescriptorSize</i>	A pointer to the location in which firmware returns the size, in bytes, of an individual <b>EFI_MEMORY_DESCRIPTOR</b> .
<i>DescriptorVersion</i>	A pointer to the location in which firmware returns the version number associated with the <b>EFI_MEMORY_DESCRIPTOR</b> . See “Related Definitions”.

## Related Definitions

```

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

```

**Type**                      Type of the memory region.

**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_WP          0x0000000000000100
#define EFI_MEMORY_RP          0x0000000000000200
#define EFI_MEMORY_XP          0x0000000000000400
#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.

<b>EFI_MEMORY_WP</b>	Physical memory protection attribute: Memory region is write-protected by system hardware.
<b>EFI_MEMORY_RP</b>	Physical memory protection attribute: Memory region is read-protected by system hardware.
<b>EFI_MEMORY_XP</b>	Physical memory protection attribute: Memory region is protected against executing code by system hardware.
<b>EFI_MEMORY_RUNTIME</b>	Runtime memory attribute: The memory region needs to be given a virtual mapping by the operating system when <b>SetVirtualAddressMap()</b> 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. Before calling **ExitBootServices()**, the executing EFI Image can use **GetMemoryMap()** to verify that the memory map supports booting from the current boot option.

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 <i>MemoryMap</i> buffer.
EFI_BUFFER_TOO_SMALL	The <i>MemoryMap</i> buffer was too small. The current buffer size needed to hold the memory map is returned in <i>MemoryMapSize</i> .
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

<i>PoolType</i>	The type of pool to allocate. The only supported types are <b>EfiLoaderData</b> , <b>EfiBootServicesData</b> , and <b>RuntimeServicesData</b> . Type <b>EFI_MEMORY_TYPE</b> is defined in Section 3.2.1.
<i>Size</i>	The number of bytes to allocate from the pool.
<i>Buffer</i>	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	<i>PoolType</i> 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	<i>Buffer</i> 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 3-6.

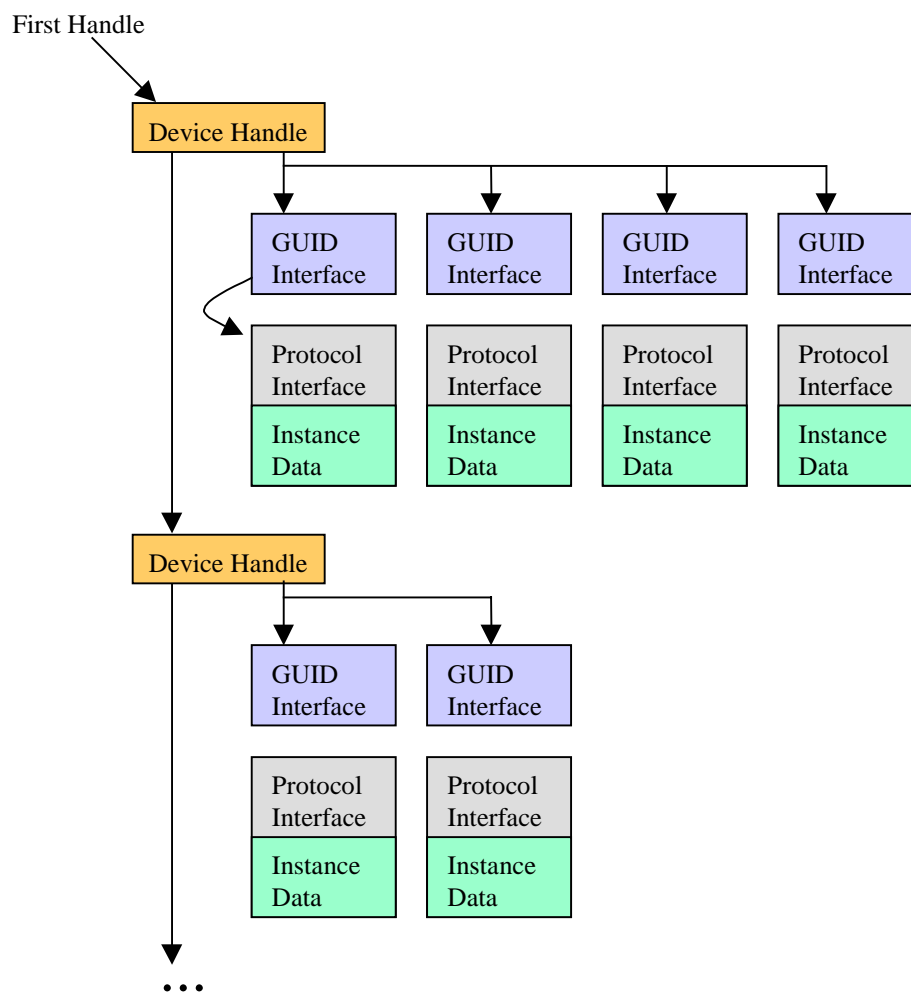
**Table 3-6. Protocol Interface Functions**

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.

1. 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

<i>Handle</i>	A pointer to the <b>EFI_HANDLE</b> on which the interface is to be installed. If <b>NULL</b> on input, a new handle is created and returned on output. If not <b>NULL</b> on input, the protocol is added to the handle, and the handle is returned unmodified. The type <b>EFI_HANDLE</b> is defined in "Related Definitions".
<i>Protocol</i>	The numeric ID of the protocol interface. The type <b>EFI_GUID</b> is defined in "Related Definitions".
<i>InterfaceType</i>	Indicates whether <i>Interface</i> is supplied in native or p-code form. This value indicates the original execution environment of the request. See "Related Definitions".
<i>Interface</i>	A pointer to the protocol interface. The <i>Interface</i> must adhere to the structure defined by <i>Protocol</i> .

## 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 notifies any process that has registered to wait for its installation. 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

<i>Handle</i>	A pointer to the handle on which the interface was installed. Type <b>EFI_HANDLE</b> is defined in Section 3.3.1.
<i>Protocol</i>	The numeric ID of the interface. Type <b>EFI_GUID</b> is defined in Section 3.3.1.
<i>Interface</i>	A pointer to the interface.

#### Description

The **UninstallProtocolInterface()** function removes a protocol interface from a device 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

<i>Handle</i>	A pointer to the handle on which the interface is to be reinstalled. Type <b>EFI_HANDLE</b> is defined in Section 3.3.1.
<i>Protocol</i>	The numeric ID of the interface. Type <b>EFI_GUID</b> is defined in Section 3.3.1.
<i>OldInterface</i>	A pointer to the old interface.
<i>NewInterface</i>	A pointer to the new interface.

#### 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*.

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.

#### Status Codes Returned

EFI_SUCCESS	The protocol interface was installed.
EFI_NOT_FOUND	The <i>OldInterface</i> 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

<i>Protocol</i>	The numeric ID of the protocol for which the event is to be registered. Type <b>EFI_GUID</b> is defined in Section 3.3.1.
<i>Event</i>	Event that is to be signaled whenever a protocol interface is registered for <i>Protocol</i> . Type <b>EFI_EVENT</b> is defined in Section 3.1.1.
<i>Registration</i>	A pointer to a memory location to receive the registration value. This value must be saved and used by the notification function of <i>Event</i> to retrieve the list of handles that have added a protocol interface of type <i>Protocol</i> .

#### Description

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

Once *Event* has been signaled, the **LocateHandle()** function can be called to identify the newly installed 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 the media re-appears, 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

<i>SearchType</i>	Specifies which handle(s) are to be returned. Type <b>EFI_LOCATE_SEARCH_TYPE</b> is defined in “Related Definitions”.
<i>Protocol</i>	Specifies the protocol to search by. This parameter is only valid if <i>SearchType</i> is <b>ByProtocol</b> . Type <b>EFI_GUID</b> is defined in Section 3.3.1.
<i>SearchKey</i>	Specifies the search key. This parameter is ignored if <i>SearchType</i> is <b>AllHandles</b> or <b>ByProtocol</b> . If <i>SearchType</i> is <b>ByRegisterNotify</b> , the parameter must be the <i>Registration</i> value returned by function <b>RegisterNotifyProtocol()</b> .
<i>BufferSize</i>	On input, the size in bytes of <i>Buffer</i> . On output, the size in bytes of the array returned in <i>Buffer</i> (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).
<i>Buffer</i>	The buffer in which the array is returned. Type <b>EFI_HANDLE</b> is defined in Section 3.3.1.

## Related Definitions

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

<b>AllHandles</b>	<i>Protocol</i> and <i>SearchKey</i> are ignored and the function returns an array of every handle in the system.
<b>ByRegisterNotify</b>	<i>SearchKey</i> supplies the <i>Registration</i> value returned by <b>RegisterProtocolNotify()</b> . 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. <i>Protocol</i> is ignored for this search type.
<b>ByProtocol</b>	All handles that support <i>Protocol</i> are returned. <i>SearchKey</i> 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 <i>BufferSize</i> is too small for the result. <i>BufferSize</i> 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

<i>Handle</i>	The device being queried. Type <b>EFI_HANDLE</b> is defined in <a href="#">Section 3.3.1</a> .
<i>Protocol</i>	The published unique identifier of the protocol. Type <b>EFI_GUID</b> is defined in <a href="#">Section 3.3.1</a> .
<i>Interface</i>	Supplies the address where a pointer to the corresponding Protocol Interface is returned.

#### 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 all devices on a device path that support a specified protocol, and returns the handle to the device that is closest to the path.

#### Prototype

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

#### Parameters

<i>Protocol</i>	The protocol to search for. Type <b>EFI_GUID</b> is defined in Section 3.3.1.
<i>DevicePath</i>	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 <b>EFI_DEVICE_PATH</b> is defined in “Related Definitions”.
<i>Device</i>	A pointer to the returned device handle. Type <b>EFI_HANDLE</b> 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 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

There are three types of images that may be loaded: EFI application, EFI Boot Services Driver, and EFI Runtime Services Driver. 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 each is loaded by the firmware's loader. Table 3-7 summarizes the differences between images.

**Table 3-7. Image Type Differences Summary**

EFI application	<p>A transient application that is loaded during boot services time. These applications are either unloaded when they complete, or they take responsibility for the continued operation of the system via <b>ExitBootServices()</b>. The applications are loaded in sequential order by the boot manager, but one application may dynamically load another.</p> <p>Loaded into <b>EfiLoaderCode</b> and <b>EfiLoaderData</b> memory.</p> <p>Default pool allocations are from <b>EfiLoaderData</b> memory.</p> <p>When an Application image exits, the firmware frees the memory used to hold the application image.</p> <p>This type of image would not install any protocol interfaces or handles.</p>
Boot Services Driver	<p>A program that is loaded into boot services memory and stays resident until boot services terminates.</p> <p>Loaded into <b>EfiBootServicesCode</b> and <b>EfiBootServicesData</b> memory.</p> <p>Default pool allocations are from <b>EfiBootServicesData</b> memory.</p> <p>When a boot services driver exits with an error code, the firmware frees the memory used to hold the driver image.</p> <p>When a boot services driver's entry point completes with <b>EFI_SUCCESS</b>, the image is retained in memory. This type of image would typically use <b>InstallProtocolInterface()</b>.</p>
Runtime Services Driver	<p>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 <b>EfiRuntimeServicesData</b>. (Note that the memory only stays resident when booting an EFI-compatible operating system. Legacy operating systems will reuse the memory).</p> <p>Default pool allocations are from <b>EfiRuntimeServicesData</b> memory.</p> <p>When a runtime service driver exits with an error code, the firmware frees the memory used to hold the driver image.</p> <p>A runtime driver can only allocate runtime memory during boot service time.</p> <p>When a boot services driver's entry point completes with <b>EFI_SUCCESS</b>, the image is retained in memory.</p>

Most images are loaded by the firmware's boot manager component. 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 3-8 lists the Image Functions:

**Table 3-8. 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

<i>BootPolicy</i>	If <b>TRUE</b> , indicates that the request originates from the boot manager, and that the boot manager is attempting to load <i>FilePath</i> as a boot selection.
<i>ParentImageHandle</i>	The caller's image handle. Type <b>EFI_HANDLE</b> is defined in Section 3.3.1.
<i>FilePath</i>	The <i>DeviceHandle</i> specific file path that the image is loaded from. Type <b>EFI_DEVICE_PATH</b> is defined in Section 3.3.7.
<i>SourceBuffer</i>	If present, the memory location where a copy of a loadable image has been put in memory.
<i>SourceSize</i>	The size in bytes of the <i>SourceBuffer</i> . This field is ignored if <i>SourceBuffer</i> is <b>NULL</b> .
<i>ImageHandle</i>	The address to store a handle that is created when the image is successfully loaded. Type <b>EFI_HANDLE</b> is defined in Section 3.3.1.

#### Description

On invoking **LoadImage()** the firmware will interpret the source data as an EFI image and load it into the proper memory addresses and apply any relocation “fixups”. To access the source data, the firmware first uses a *SourceAddress* and *SourceSize* if present. In this case, the caller has copied the source data into memory and the load operation is a memory to memory load. Once loaded the caller may free the *SourceBuffer*. If *SourceAddress* is **NULL**, the firmware attempts to use the **SIMPLE\_FILE\_SYSTEM** protocol and then the **LOAD\_FILE\_PROTOCOL** on the *DeviceHandle* to access the file referred to by *FilePath*.

The *BootPolicy* flag is passed to the **LOAD\_FILE\_PROTOCOL** interface and is used to load the default image responsible for booting when the *FilePath* only indicates the device. For more information concerning this settings see the **LOAD\_FILE\_PROTOCOL**.

Once the image is loaded, an **EFI\_HANDLE** is created to identify the image that 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 with **StartImage()**.

Once the image is loaded, the caller must either start the image with **StartImage()** or unload the image by calling **Exit()**.

## Status Codes Returned

EFI_SUCCESS	Image was loaded into memory correctly.
EFI_NOT_FOUND	The <i>FilePath</i> was not found.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_OUT_OF_RESOURCES	Image was not loaded because a resource ran out.
EFI_LOAD_ERROR	Image was not loaded because the image format was not understood or corrupt.
EFI_DEVICE_ERROR	Image was not loaded because a read to the device returned an 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

<i>ImageHandle</i>	Identifies which image to start. Type <b>EFI_HANDLE</b> is defined in Section 3.3.1.
<i>ExitDataSize</i>	The location to return the size of the returned <i>ExitData</i> buffer.
<i>ExitData</i>	The location to return 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

This function transfers control to the image's entry point, which was loaded by **LoadImage()**. The image may only be started one time.

The return from **StartImage()** is caused by the loaded image calling **Exit()**. The *ExitData* buffer from **Exit()** is passed back through the *ExitData* buffer. The size of the *ExitData* buffer is passed back through *ExitDataSize*. The caller is responsible for freeing the *ExitData* buffer back to pool with **FreePool()** when the buffer is no longer needed.

#### Status Codes Returned

EFI_INVALID_PARAMETER	<i>ImageHandle</i> 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

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

#### Parameters

*ImageHandle* A handle that identifies the loaded image. Type **EFI\_HANDLE** is defined in [Section 3.3.1](#).

#### Description

If the image referred to by *ImageHandle* has not been started, the image is unloaded.

If the image has been started and the image has supplied an **Unload()** entry point control is passed to the image's unload entry point. If the image's unload function returns success, the image is unloaded; otherwise, the error is returned to the caller. The image 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** from the unload operation.

#### 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	One of the parameters has an invalid value.
Exit code from Unload handler	Exit code from image's unload handler.

### 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) (
    EFI_HANDLE                *ImageHandle
    IN EFI_SYSTEM_TABLE       *SystemTable
);
```

#### Parameters

<i>ImageHandle</i>	A handle that identifies the loaded image. Type <b>EFI_HANDLE</b> is defined in Section 3.3.1.
<i>SystemTable</i>	The System Table for this image. Type <b>EFI_SYSTEM_TABLE</b> is defined in Chapter 4.

#### Description

An image's entry point is of type **EFI\_IMAGE\_ENTRY\_POINT**. After the firmware loads the image to 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 binding and data information the image may need. This information is available through protocol interfaces. However, to access the protocol interfaces on *ImageHandle* requires access to the boot services. Therefore, **LoadImage()** passes a *SystemTable* to the **EFI\_IMAGE\_ENTRY\_POINT** that is inherited from the current scope of **LoadImage()**.

All image handles support the **LOADED\_IMAGE\_PROTOCOL**. This protocol can be used to determine state about the loaded image, such as the device the image was loaded from and the load options for the image. In addition, the *ImageHandle* may support other protocols provided by the parent image.

In general an image returns control from its initialization entry point by calling **Exit()**. If the image returns 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.

### 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           DataSize,
    IN CHAR16          *ExitData OPTIONAL
);
```

#### Parameters

<i>ImageHandle</i>	Identifies which image is exiting. This parameter is passed to the image on entry. Type <b>EFI_HANDLE</b> is defined in Section 3.3.1.
<i>ExitStatus</i>	The exit code of the currently executing EFI Image.
<i>DataSize</i>	The size of <i>ExitData</i> in bytes.
<i>ExitData</i>	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. <i>ExitData</i> is only valid if <i>ExitStatus</i> is something other than <b>EFI_SUCCESS</b> . The <i>ExitData</i> buffer must be allocated via <b>AllocatePool()</b> .

#### Description

An EFI Image can call this function only if it is the current image that the firmware has transferred execution control to. If the image has already returned from its **EFI\_IMAGE\_ENTRY** point, then this function may not be called. Also, if the image has loaded one or more child images, this function cannot be called until the child images have exited. Using **Exit()** is similar to returning from the Image's **EFI\_IMAGE\_ENTRY** point with the difference that this function may also return additional *ExitData*.

When an EFI application exits, the memory used to hold the image is always freed back to available memory, and the firmware's references to the *ImageHandle* are freed, causing the handle to be freed. The application is responsible for freeing all resources it may have allocated before exiting. This would include any allocated memory (pages and/or pool), open file system handles, etc. The only exception to this is the *ExitData* buffer. If passing an *ExitData* buffer, the buffer must be allocated via **AllocatePool()** and is provided to the caller of **StartImage()**. The caller of **StartImage()** is responsible for freeing the *ExitData* buffer.

When a boot service driver or runtime service driver exits, the image is freed only if the *ExitStatus* is an error code; otherwise the image stays resident in memory. The driver must not return an error status 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 would include any allocated memory (pages and/or pool), open file system handles, etc.

It is valid to call **Exit()** for an image that was loaded with **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 <b>StartImage()</b> call that invoked the image.
EFI_SUCCESS	The image was unloaded. <b>Exit()</b> only returns success if the image has not been started; otherwise, the exit returns to the <b>StartImage()</b> call that invoked the image.
EFI_INVALID_PARAMETER	The requested image is not the current image that can exit.

### 3.4.6 ExitBootServices()

#### Summary

Terminates all boot services.

#### Prototype

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

#### Parameters

*ImageHandle* Identifies which image is exiting boot services. Type **EFI\_HANDLE** is defined in Section 3.3.1.

*MapKey* The 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 EFI OS 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 insure that the memory map does not change between obtaining the key and calling **ExitBootServices()**. It is suggested that the EFI OS loader calls **GetMemoryMap()** immediately before calling **ExitBootServices()**.

On success, the EFI OS loader owns all available memory in the system. In addition, all memory in the map marked as **EfiBootServicesCode** and **BootServiceData** can be treated by the EFI OS loader as available free memory. No further calls to boot service interfaces or EFI device handle derived protocol services may be used. The Boot Services watchdog timer is disabled.

#### Status Codes Returned

EFI_SUCCESS	Boot services have been terminated.
EFI_INVALID_PARAMETER	The 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 OS loaders and other applications that run in the EFI environment.

Although the implementation of variable storage is not defined in this specification, they are required to 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 3-9 lists the variable services functions described in this section:

**Table 3-9. 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

<i>VariableName</i>	A Null-terminated Unicode string that is the name of the vendor's variable.
<i>VendorGuid</i>	A unique identifier for the vendor. Type <b>EFI_GUID</b> is defined in Section 3.3.1.
<i>Attributes</i>	If not <b>NULL</b> , a pointer to the memory location to return the attributes bitmask for the variable. See “Related Definitions”.
<i>DataSize</i>	On input, the size in bytes of the return <i>Data</i> buffer. On output the size of data returned in <i>Data</i> .
<i>Data</i>	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, at runtime, 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 <i>BufferSize</i> is too small for the result. <i>BufferSize</i> 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

<i>VariableNameSize</i>	The size of the <i>VariableName</i> buffer.
<i>VariableName</i>	On input, supplies the last <i>VariableName</i> that was returned by <b>GetNextVariableName()</b> . On output, returns the Null-terminated Unicode string of the current variable.
<i>VendorGuid</i>	On input, supplies the last <i>VendorGuid</i> that was returned by <b>GetNextVariableName()</b> . On output, returns the <i>VendorGuid</i> of the current variable. Type <b>EFI_GUID</b> 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 would typically be done on the initial call to **GetNextVariableName()**.

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 <i>VariableNameSize</i> is too small for the result. <i>VariableNameSize</i> 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

<i>VariableName</i>	A Null-terminated Unicode string that is the name of the vendor's variable. Each <i>VariableName</i> is unique for each <i>VendorGuid</i> .
<i>VendorGuid</i>	A unique identifier for the vendor. Type <b>EFI_GUID</b> is defined in Section 3.3.1.
<i>Attributes</i>	Attributes bitmask to set for the variable. See Section 3.5.1.
<i>DataSize</i>	The size in bytes of the <i>Data</i> buffer. A size of zero causes the variable to be deleted.
<i>Data</i>	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.

Storage for **EFI\_VARIABLE\_NON\_VOLATILE** variables are done 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.

Variable must contain 1 or more bytes of *Data*. Using **SetVariable()** with a *DataSize* of zero causes the entire variable to be removed from storage.

The Attributes have the following usage rules:

- Storage attributes are only applied to a variable when creating the variable. If the same variable is added to different variable stores without first deleting the original variable, the firmware may delete the new variable on the next power cycle.
- 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 the variable store offers no read, write, or delete security protection.

The size of the *VariableName*, including the Unicode Null in bytes plus the *DataSize* is limited to a maximum size of 1024 bytes.

## 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 <i>VariableSize</i> 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 3-10 lists the time services functions described in this section:

**Table 3-10. 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

<i>Time</i>	A pointer to storage to receive a snapshot of the current time. Type <b>EFI_TIME</b> is defined in “Related Definitions”.
<i>Capabilities</i>	An optional pointer to a buffer to receive the real time clock device’s capabilities. Type <b>EFI_TIME_CAPABILITIES</b> is defined in “Related Definitions”.

#### Related Definitions

```

//*****
//EFI_LOCAL_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_LOCAL_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.nnnnnnnnnn*. 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 interrupted as a local time.

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

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

If **EFI\_TIME\_IN\_DAYLIGHT** is set, the time has been adjusted for daylight savings time.

All other bits must be zero.

```

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

```

*Accuracy* Provides the timekeeping accuracy of the real-time clock in an error rate of 1E-6 parts per million. For a clock using a standard uncompensated crystal the value is 50 parts per million (or 50,000,000).

*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.

*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 0.

### 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. E.g., 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

<i>Enabled</i>	Indicates if the alarm is currently enabled or disabled.
<i>Pending</i>	Indicates if the alarm signal is pending and requires acknowledgement.
<i>Time</i>	The current alarm setting. Type <b>EFI_TIME</b> is defined in Section 0.

#### 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 0. 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	The alarm was set.
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 3-11 lists the virtual memory service functions described in this section:

**Table 3-11. 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

<i>MemoryMapSize</i>	The size in bytes of <i>VirtualMap</i> .
<i>DescriptorSize</i>	The size in bytes of an entry in the <i>VirtualMap</i> .
<i>DescriptorVersion</i>	The version of the structure entries in <i>VirtualMap</i> .
<i>VirtualMap</i>	An array of memory descriptors which contain new virtual address mapping information for all runtime ranges. Type <b>EFI_MEMORY_DESCRIPTOR</b> 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 supply the new virtual address for every entry in the memory map that was in effect at **ExitBootServices()** time that were 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	The supplied <i>VirtualMap</i> does not contain virtual address mapping information for all the runtime components. No changes were applied.

### 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

<i>DebugDisposition</i>	Supplies type information for the pointer being converted. See Related Definitions.
<i>Address</i>	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, the EFI firmware fixes any compiled in pointers that were imbedded in any runtime image.

#### Status Codes Returned

EFI_SUCCESS	The pointer pointed to by <i>Address</i> was modified.
EFI_NOT_FOUND	The pointer pointed to by <i>Address</i> 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 3-12 lists the Miscellaneous Services Functions.

**Table 3-12. 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.

### 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

<i>ResetType</i>	The type of reset to perform. Type <b>EFI_RESET_TYPE</b> is defined in “Related Definitions”.
<i>ResetStatus</i>	The status code for the reset. If the system reset is part of a normal operation, the status code would be <b>EFI_SUCCESS</b> . If the system reset is due to some type of failure the most appropriate EFI Status code would be used.
<i>DataSize</i>	The size, in bytes, of <i>ResetData</i> .
<i>ResetData</i>	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. <i>ResetData</i> is only valid if <i>ResetStatus</i> is something other than <b>EFI_SUCCESS</b> . 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

<i>Timeout</i>	The number of seconds to set the watchdog timer to. A value of zero disables the timer.
<i>WatchdogCode</i>	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.
<i>DataSize</i>	The size, in bytes, of <i>WatchdogData</i> .
<i>WatchdogData</i>	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. *Just before firmware calls the EFI Image's entry point, the watchdog timer is set for a timeout of 5 minutes.*

The EFI Image may reset or disable the watchdog timer as needed.

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 <i>WatchdogCode</i> 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 <i>Microseconds</i> .
-------------	--



### 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 than 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 32bits. 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 **GetNextMonotonicCount()**] 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.

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

<i>Revision</i>	Defines the revision of the <b>EFI_LOADED_IMAGE</b> structure. All future revisions will be backward compatible to the current revision.
<i>ParentHandle</i>	Parent image's image handle. <b>NULL</b> if the image is loaded directly from the firmware's boot manager. Type <b>EFI_HANDLE</b> is defined in Chapter 3.
<i>SystemTable</i>	The image's EFI system table pointer. Type <b>EFI_SYSTEM_TABLE</b> is defined in 4.5.1.
<i>DeviceHandle</i>	The device handle that the EFI Image was loaded from. Type <b>EFI_HANDLE</b> is defined in Chapter 3.
<i>FilePath</i>	The file path portion specific to <i>DeviceHandle</i> that the EFI Image was loaded from. Type <b>EFI_DEVICE_PATH</b> is defined in Chapter 3.
<i>Reserved</i>	Reserved. DO NOT USE.
<i>LoadOptionsSize</i>	The size in bytes of <i>LoadOptions</i> .
<i>LoadOptions</i>	The image's binary load options.
<i>ImageBase</i>	The base address at which the image was loaded.
<i>ImageSize</i>	The size in bytes of the loaded image.
<i>ImageCodeType</i>	The memory type that the code sections were loaded as. Type <b>EFI_MEMORY_TYPE</b> is defined in Chapter 3.
<i>ImageDataType</i>	The memory type that the data sections were loaded as. Type <b>EFI_MEMORY_TYPE</b> is defined in Chapter 3.
<i>Unload</i>	Function that unloads the image. See Section 0.

## 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 <i>ImageHandle</i> 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 IA-64 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      (0<<16) | (91)

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 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.

```

// Example interface prototype

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

//
// EFI Runtime Services Table
//

#define EFI_RUNTIME_SERVICES_SIGNATURE  0x56524553544e5552
#define EFI_RUNTIME_SERVICES_REVISION  (0<<16) | (91)

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    (0<<16) | (91)

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_RESERVED_SERVICE              ReservedService0;

    //
    // Protocol Handler Services
    //

    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_RESERVED_SERVICE              ReservedService1;

    //
    // Image Services
    //

```

```

EFI_IMAGE_LOAD          LoadImage;
EFI_IMAGE_START         StartImage;
EFI_EXIT                Exit;
EFI_IMAGE_UNLOAD        UnloadImage;
EFI_EXIT_BOOT_SERVICES  ExitBootServices;

//
// Miscellaneous Services
//

EFI_GET_NEXT_MONOTONIC_COUNT  GetNextMonotonicCount;
EFI_STALL                    Stall;
EFI_SET_WATCHDOG_TIMER        SetWatchdogTimer;

} 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.

Stack	Location
EFI_SYSTEM_TABLE *	ESP + 8
EFI_HANDLE	ESP + 4
<return address>	ESP

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



4.5.1.2 IA-64 Handoff State

EFI uses the standard P64 C calling conventions that are defined for IA-64. Figure 4-2 shows the stack after **ImageEntryPoint** has been called on IA-64 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, IA-64

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.



## 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 media 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 \
    { 9576e91-6d3f-11d2-8e39-00a0c969723b }
```

### Protocol Interface Structure

```
EFI_DEVICE_PATH      *DevicePath;
```

### Parameters

<i>DevicePath</i>	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 <b>EFI_DEVICE_PATH</b> 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 the media 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 0; 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 Type 0x02 – ACPI Device Path Type 0x03 – Messaging Device Path Type 0x04 – Media Device Path Type 0x05 – BIOS Boot Specification Device Path Type 0xFF – End of Hardware Device Path
Sub-Type	1	1	Sub-Type – Varies by Type. (See Table 5-2.)
Length	2	2	Length of this structure in bytes. Length is 4 + <i>n</i> bytes.
Specific Device Path Data	4	<i>n</i>	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 nodes 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 0xFF – End of Hardware Device Path.
Sub-Type	1	1	Sub-Type 0xFF – End Entire Device Path, or Sub-Type 0x01 – End This Instance of a Device Path and start a new Device Path.
Length	2	2	Length of this structure in bytes. Length is 4 bytes.

### 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 8 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	<b>EFI_MEMORY_TYPE</b> (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.

### 5.3.3 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 ACPI 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 5-7. 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 5-8. 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 5-9. 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 6 bytes.
Target ID	4	2	Target ID on the SCSI bus, PUN
Logical Unit Number	5	2	Logical Unit Number, LUN

### 5.3.4.3 Fibre Channel Device Path

**Table 5-10. 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 16 bytes.
Reserved	4	4	Reserved
World Wide Number	8	8	Fibre Channel World Wide Number

### 5.3.4.4 1394 Device Path

**Table 5-11. 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	8	8	1394 Global Unique ID (GUID)

### 5.3.4.5 USB Device Path

**Table 5-12. 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 6 bytes.
USB Port Number	4	1	USB Port Number
Reserved	5	3	Reserved
Device Address	8	8	Device Address consists of Hub ID Number and the USB Device Address.

### 5.3.4.6 I<sub>2</sub>O Device Path

**Table 5-13. I<sub>2</sub>O Device Path**

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 6 – I <sub>2</sub> O 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.

### 5.3.4.7 MAC Address Device Path

**Table 5-14. 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 21 bytes.
MAC Address	4	16	The MAC address for a network interface padded with 0s.
IfType	20	1	Network interface type(i.e. 802.3, FDDI). See RFC 1700.

### 5.3.4.8 IPv4 Device Path

**Table 5-15. IPv4 Device Path**

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 12 – IPv4
Length	2	2	Length of this structure in bytes. Length is 19 bytes.
Local IP Address	4	4	The local IPv4 address
Remote IP Address	8	4	The remote IPv4 address
Local Port	12	2	The local port number
Remote Port	14	2	The remote port number
Protocol	16	2	The network protocol(i.e. UDP, TCP). See RFC 1700.
StaticIPAddress	18	1	0x00 - The Source IP Address was assigned though DHCP. 0x01 - The Source IP Address is statically bound.



### 5.3.4.9 IPv6 Device Path

Table 5-16. IPv6 Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 13 – IPv6
Length	2	2	Length of this structure in bytes. Length is 43 bytes.
Local IP Address	4	16	The local IPv6 address
Remote IP Address	20	16	The remote IPv6 address
Local Port	36	2	The local port number
Remote Port	38	2	The remote port number
Protocol	40	2	The network protocol (i.e. UDP, TCP). See RFC 1700.
StaticIPAddress	42	1	0x00 - The Source IP Address was assigned though DHCP. 0x01 - The Source IP Address is statically bound.

### 5.3.4.10 InfiniBand<sup>†</sup> Device Path

Table 5-17. InfiniBand Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 9 – InfiniBand
Length	2	2	Length of this structure in bytes. Length is TBD bytes.

### 5.3.4.11 UART Device Path

**Table 5-18. 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 12 bytes.
Baud Rate	4	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	9	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	10	1	The parity setting for the UART style device. Parity 0x00 - Default Parity Parity 0x01 - No Parity Parity 0x02 - Even Parity Parity 0x03 - Odd Parity Parity 0x04 - Mark Parity Parity 0x05 - Space Parity
Stop Bits	11	1	The number of stop bits for the UART style device. Stop Bits 0x00 - Default Stop Bits Stop Bits 0x01 - 1 Stop Bit Stop Bits 0x02 - 1.5 Stop Bits Stop Bits 0x03 - 2 Stop Bits

### 5.3.4.12 Vendor-Defined Messaging Device Path

Table 5-19. 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 + <i>n</i> bytes.
Vendor_GUID	4	16	Vendor-assigned GUID that defines the data that follows.
Vendor Defined Data	20	<i>n</i>	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 5-20. 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 medium.
Partition Size	16	8	Size of the partition in units of Logical Blocks.
Partition Signature	24	16	Signature unique to this partition.
MBR Type	40	1	MBR Type: (Unused values reserved) 0x01 – PC AT compatible MBR. Partition Start and Partition Size come from <code>PartitionStartingLBA</code> and <code>PartitionSizeInLBA</code> for the partition. 0x02 – EFI Partition Table Header.
Signature Type	41	1	Type of Disk Signature: (Unused values reserved) 0x00 – No Disk Signature. 0x01 – 32-bit signature from address 0x1b8 of the type 0x01 MBR. 0x02 – GUID signature.

The following structure defines 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 5-21. 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 media. 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 5-22. 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 + <i>n</i> bytes.
Vendor_GUID	4	16	Vendor-assigned GUID that defines the data that follows.
Vendor Defined Data	20	<i>n</i>	Vendor-defined variable size data.

### 5.3.5.4 File Path Media Device Path

**Table 5-23. 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	20	$n$	Unicode Path string including directory and file names. The length of this string $n$ can be determined by subtracting 20 from the Length entry.

### 5.3.5.5 Media Protocol

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. The Media Protocol device path node is only used for the protocol types listed below.

**Table 5-24. 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 5-25. 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 20 + <i>n</i> 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	20	<i>n</i>	ASCII string that describes the boot device to a user. The length of this string <i>n</i> 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 5-26. 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 5-27. ACPI \_ADR to EFI Device Path Mapping**

ACPI _ADR Bus	EFI Device Path
EISA	<i>Not supported</i>
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	<i>Not Supported</i> – May be the same as PC CARD???
PC CARD	PC CARD Hardware Device Path
SMBus	<i>Not Supported</i>

#### 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 format. 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 a legacy medium.

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.

## 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 “programmatically 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

This section provides a detailed description of the **DEVICE\_IO** protocol.

### Summary

Provides the basic Memory, I/O, and PCI interfaces that a driver uses to access its 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_DEVICE_IO_INTERFACE;
```

### Parameters

<i>Mem</i>	Allows reads and writes to PCI memory space. See Section 6.2.1.
<i>Io</i>	Allows reads and writes to PCI I/O space. See Section 6.2.1.
<i>Pci</i>	Allows reads and writes to PCI configuration space. See Section 6.2.1.
<i>Map</i>	Provides the device specific addresses needed to access host memory for DMA.
<i>PciDevicePath</i>	Provides an EFI Device Path for a PCI device with the given PCI configuration space address.
<i>Unmap</i>	Releases any resources allocated by Map.
<i>AllocateBuffer</i>	Allocates pages that are suitable for a common buffer mapping.
<i>Flush</i>	Flushes any posted write data to the device.

## Description

The **DEVICE\_IO** protocol provides the basic Memory, I/O, and PCI interfaces that a driver uses to access its 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

<i>This</i>	The <b>EFI_DEVICE_IO_INTERFACE</b> instance. Type <b>EFI_DEVICE_IO_INTERFACE</b> is defined in Section 6.2.
<i>Width</i>	Signifies the width of the I/O operations. Type <b>EFI_IO_WIDTH</b> is defined in Section 6.2.
<i>Address</i>	The base address of the I/O operations. The caller is responsible for aligning the <i>Address</i> if required.
<i>Count</i>	The number of I/O operations to perform. Bytes moved is <i>Width</i> size * <i>Count</i> , starting at <i>Address</i> .
<i>Buffer</i>	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 **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.
Reserved	4	4	Must be zero.

### Status Codes Returned

EFI_SUCCESS	The data was read from or written to the device.
EFI_UNSUPPORTED	The <i>Address</i> is not valid for this system.
EFI_INVALID_PARAMETER	<i>Width</i> or <i>Count</i> , 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

<i>This</i>	The <b>EFI_DEVICE_IO_INTERFACE</b> . Type <b>EFI_DEVICE_IO_INTERFACE</b> is defined in Section 6.2.
<i>PciAddress</i>	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.
<i>PciDevicePath</i>	A pointer to the pointer for the EFI Device Path for <i>PciAddress</i> . Memory for the Device Path is allocated from the pool. Type <b>EFI_DEVICE_PATH</b> 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 <i>PciDevicePath</i> returns a pointer to a valid EFI Device Path.
EFI_UNSUPPORTED	The <i>PciAddress</i> 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 host 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

<i>This</i>	The <b>EFI_DEVICE_IO_INTERFACE</b> instance. Type <b>EFI_DEVICE_IO_INTERFACE</b> is defined in Section 6.2.
<i>Operation</i>	Indicates if the bus master is going to read or write to host memory. Type <b>EFI_IO_OPERATION_TYPE</b> is defined in “Related Definitions”.
<i>HostAddress</i>	The host memory address to map to the device. Type <b>EFI_PHYSICAL_ADDRESS</b> is defined in Chapter 3.
<i>NumberOfBytes</i>	On input the number of bytes to map. On output the number of bytes that were mapped.
<i>DeviceAddress</i>	The resulting map address for the bus master device to use to access the hosts <i>HostAddress</i> . Type <b>EFI_PHYSICAL_ADDRESS</b> is defined in Chapter 3.
<i>Mapping</i>	A resulting value to pass to <b>Unmap ( )</b> .

#### Related Definitions

```
typedef enum {
    EFIBusMasterRead,
    EFIBusMasterWrite,
    EFIBusMasterCommonBuffer
} EFI_IO_OPERATION_TYPE;
```

## Description

The **DEVICE\_IO.Map()** function provides the device specific addresses needed to access host memory. This function is used to map host 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 host memory before the **Map()** is performed. Similarly, if performing an **EFIBusMasterWrite**, the data can not be properly accessed in host 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 <i>NumberOfBytes</i> .
EFI_INVALID_PARAMETER	The <i>Operation</i> or <i>HostAddress</i> is undefined.
EFI_UNSUPPORTED	The <i>HostAddress</i> 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*                      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 host 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 host 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

<i>This</i>	The <b>EFI_DEVICE_IO_INTERFACE</b> . Type <b>EFI_DEVICE_IO_INTERFACE</b> is defined in Section 6.2.
<i>Type</i>	The type allocation to perform. Type <b>EFI_ALLOCATE_TYPE</b> is defined in Chapter 3.
<i>MemoryType</i>	The type of memory to allocate, <b>EfiBootServicesData</b> or <b>EfiRuntimeServicesData</b> . Type <b>EFI_MEMORY_TYPE</b> is defined in Chapter 3.
<i>Pages</i>	The number of pages to allocate.
<i>HostAddress</i>	A pointer to store the base address of the allocated range. Type <b>EFI_PHYSICAL_ADDRESS</b> 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 **FreePages()**.

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.

## 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*                      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.





## 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 equivalence 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.
0x15	Function 11.
0x16	Function 12.
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

<i>Reset</i>	Reset the <i>ConsoleIn</i> device. See Section 7.3.1.
<i>ReadKeyStroke</i>	Returns the next input character. See Section 7.3.2.
<i>WaitForKey</i>	Event to use with <b>WaitForEvent()</b> 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
(EFI_API *EFI_INPUT_RESET) (
    IN SIMPLE_INPUT_INTERFACE  *This
    IN BOOLEAN                  ExtendedVerification
);
```

#### Parameters

<i>This</i>	The <b>SIMPLE_INPUT_INTERFACE</b> instance. Type <b>SIMPLE_INPUT_INTERFACE</b> is defined in Section 0.
<i>ExtendedVerification</i>	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
(EFI_API *EFI_INPUT_READ_KEY) (
    IN SIMPLE_INPUT_INTERFACE  *This,
    OUT EFI_INPUT_KEY          *Key
);
```

#### Parameters

<i>This</i>	The <b>SIMPLE_INPUT_INTERFACE</b> instance. Type <b>SIMPLE_INPUT_INTERFACE</b> is defined in Section 0.
<i>Key</i>	A pointer to a buffer that is filled in with the keystroke information for the key that was pressed. Type <b>EFI_INPUT_KEY</b> 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

<i>Reset</i>	Reset the <i>ConsoleOut</i> device. See Section 7.5.1.
<i>OutputString</i>	Displays the Unicode string on the device at the current cursor location. See Section 7.5.2.
<i>TestString</i>	Tests to see if the <i>ConsoleOut</i> device supports this Unicode string. See Section 0.
<i>QueryMode</i>	Queries information concerning the output device's supported text mode. See Section 0.
<i>SetMode</i>	Sets the current mode of the output device. See Section 7.5.5.

<i>SetAttribute</i>	Sets the foreground and background color of the text that is output. See Section 7.5.6.
<i>ClearScreen</i>	Clears the screen with the currently set background color. See Section 0.
<i>SetCursorPosition</i>	Sets the current cursor position. See Section 0.
<i>EnableCursor</i>	Turns the visibility of the cursor on/off. See Section 7.5.9.
<i>Mode</i>	Pointer to <b>SIMPLE_TEXT_OUTPUT_MODE</b> data. Type <b>SIMPLE_TEXT_OUTPUT_MODE</b> 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:

<i>MaxMode</i>	The number of modes supported by QueryMode() and SetMode().
<i>Mode</i>	The text mode of the output device(s).
<i>Attribute</i>	The current character output attribute.
<i>CursorColumn</i>	The cursor’s column.
<i>CursorRow</i>	The cursor’s row.
<i>CursorVisible</i>	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* 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

<i>This</i>	The <b>SIMPLE_TEXT_OUTPUT_INTERFACE</b> instance. Type <b>SIMPLE_TEXT_OUTPUT_INTERFACE</b> is defined in Section 7.5.
<i>String</i>	The Null-terminated Unicode string to be displayed on the output device(s).

### 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
(EFI_API *EFI_TEXT_TEST_STRING) (
    IN SIMPLE_TEXT_OUTPUT_INTERFACE  *This,
    IN CHAR16                        *String
);
```

#### Parameters

<i>This</i>	The <b>SIMPLE_TEXT_OUTPUT_INTERFACE</b> instance. Type <b>SIMPLE_TEXT_OUTPUT_INTERFACE</b> is defined in Section 7.5.
<i>String</i>	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 device(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
(EFI_API *EFI_TEXT_QUERY_MODE) (
    IN SIMPLE_TEXT_OUTPUT_INTERFACE *This,
    IN UINTN ModeNumber,
    OUT UINTN *Columns,
    OUT UINTN *Rows
);
```

### Parameters

*This* 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
(* EFI_API EFI_TEXT_SET_MODE) (
    IN SIMPLE_TEXT_OUTPUT_INTERFACE *This,
    IN UINTN ModeNumber
);
```

#### Parameters

*This* 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*                      The **SIMPLE\_TEXT\_OUTPUT\_INTERFACE** instance. Type **SIMPLE\_TEXT\_OUTPUT\_INTERFACE** is defined in Section 7.5.

*Attribute*                The attribute to set. 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_YELLOW     0x06
#define EFI_WHITE      0x07
#define EFI_BRIGHT     0x08

#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
(EFI_API *EFI_TEXT_CLEAR_SCREEN) (
    IN SIMPLE_TEXT_OUTPUT_INTERFACE *This
);
```

### Parameters

*This* 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* 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
(EFI_API *EFI_TEXT_ENABLE_CURSOR) (
    IN SIMPLE_TEXT_OUTPUT_INTERFACE *This,
    IN BOOLEAN                      Visible
);
```

### Parameters

<i>This</i>	The <b>SIMPLE_TEXT_OUTPUT_INTERFACE</b> instance. Type <b>SIMPLE_TEXT_OUTPUT_INTERFACE</b> is defined in Section 7.5.
<i>Visible</i>	If <b>TRUE</b> , the cursor is set to be visible. If <b>FALSE</b> , 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.

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

<i>Revision</i>	The revision to which the block IO interface adheres. All future revisions must be backwards compatible. If a future version is not backwards compatible it is not the same GUID.
<i>Media</i>	A pointer to the <b>EFI_BLOCK_IO_MEDIA</b> data for this device. Type <b>EFI_BLOCK_IO_MEDIA</b> is defined in “Related Definitions”.
<i>Reset</i>	Resets the block device hardware. See Section 8.1.1.
<i>ReadBlocks</i>	Reads the requested number of blocks from the device. See Section 8.1.2.
<i>WriteBlocks</i>	Writes the requested number of blocks to the device. See Section 0.
<i>FlushBlocks</i>	Flushes and cache blocks. This function is optional and only needs to be supported on block devices that cache writes. See Section 0.

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:

<i>MediaId</i>	The current MediaId. If the media changes, the MediaId value is changed.
<i>RemovableMedia</i>	<b>TRUE</b> if the media is removable media; otherwise, <b>FALSE</b> .
<i>MediaPresent</i>	<b>TRUE</b> if there is media currently present in the device; otherwise, <b>FALSE</b> .
<i>LogicalPartition</i>	<b>TRUE</b> if LBA 0 is the first block of a partition; otherwise <b>FALSE</b> . For media with only one partition this would be <b>TRUE</b> .
<i>ReadOnly</i>	<b>TRUE</b> if the media is marked read-only.
<i>WriteCaching</i>	<b>TRUE</b> if the <b>WriteBlock()</b> function caches write data.
<i>BlockSize</i>	The intrinsic block size of the device.
<i>IoAlign</i>	Supplies the alignment requirement for any buffer to read or write block(s).
<i>LastBlock</i>	The last logical block address on the device.

## 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 medium. The firmware is responsible for adding device handles for each partition on such media.

The firmware is responsible for adding a **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
(EFI_API *EFI_BLOCK_RESET) (
    IN EFI_BLOCK_IO          *This,
    IN BOOLEAN                ExtendedVerification
);
```

#### Parameters

<i>This</i>	Indicates the calling context. Type <b>EFI_BLOCK_IO</b> is defined in Section 8.1.
<i>ExtendedVerification</i>	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 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
(EFI_API *EFI_BLOCK_READ) (
    IN EFI_BLOCK_IO      *This,
    IN UINT32             MediaId,
    IN EFI_LBA            LBA,
    IN UINTN              BufferSize,
    OUT VOID              *Buffer
);
```

#### Parameters

<i>This</i>	Indicates the calling context. Type <b>EFI_BLOCK_IO</b> is defined in Section 8.1.
<i>MediaId</i>	The media id that the read request is for.
<i>LBA</i>	The starting logical block address to read from on the device. Type <b>EFI_LBA</b> is defined in Section 8.1.
<i>BufferSize</i>	The size of the <i>Buffer</i> in bytes. This must be a multiple of the intrinsic block size of the device.
<i>Buffer</i>	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 <i>MediaId</i> is not for the current media.
EFI_BAD_BUFFER_SIZE	The <i>BufferSize</i> 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

<i>This</i>	Indicates the calling context. Type <b>EFI_BLOCK_IO</b> is defined in Section 8.1.
<i>MediaId</i>	The media id that the write request is for.
<i>LBA</i>	The starting logical block address to be written. The caller is responsible for writing to only legitimate locations. Type <b>EFI_LBA</b> is defined in Section 8.1.
<i>BufferSize</i>	The size in bytes of <i>Buffer</i> . This must be a multiple of the intrinsic block size of the device.
<i>Buffer</i>	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 <i>MediaId</i> 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 <i>BufferSize</i> 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 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.



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_BLOCK_IO           *BlkIo;
    EFI_DISK_READ          ReadDisk;
    EFI_DISK_WRITE         WriteDisk;
} EFI_DISK_IO;
```

### Parameters

<i>Revision</i>	The revision to which the disk I/O interface adheres.
<i>BlkIo</i>	A pointer to the block I/O interface that the <b>EFI_DISK_IO</b> interface is using. Type <b>EFI_BLOCK_IO</b> is defined in Chapter 8.
<i>ReadDisk</i>	Reads data from the disk. See Section 9.1.1.
<i>WriteDisk</i>	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 *BlkIo* device.

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

<i>This</i>	Indicates the calling context. Type <b>EFI_DISK_IO</b> is defined in Section 9.1.
<i>MediaId</i>	The media id that the read request is for.
<i>Offset</i>	The starting byte offset on the logical block I/O device to read from.
<i>BufferSize</i>	The size in bytes of <i>Buffer</i> . The number of bytes to read from the device.
<i>Buffer</i>	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 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 performing the read operation.
EFI_NO_MEDIA	There is no media in the device.
EFI_MEDIA_CHANGED	The <i>MediaId</i> is not for the current media.
EFI_INVALID_PARAMETER	The read request contains devices 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 UINTN             BufferSize,
    OUT VOID             *Buffer
);
```

### Parameters

<i>This</i>	Indicates the calling context. Type <b>EFI_DISK_IO</b> is defined in Section 9.1.
<i>MediaId</i>	The media id that the write request is for.
<i>Offset</i>	The starting byte offset on the logical block I/O device to write.
<i>BufferSize</i>	The size in bytes of <i>Buffer</i> . The number of bytes to write to the device.
<i>Buffer</i>	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 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 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 <i>MediaId</i> is not for the current media.
EFI_DEVICE_ERROR	The device reported an error while performing the write operation.
EFI_INVALID_PARAMETER	The write request contains devices addresses that are not valid for the device.

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\_HANDLE** 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 \  
    { 964e5b22-6459-11d2-8e39-00a0c969723b }
```

### 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.

*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\_HANDLE** 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.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_HANDLE         **Root
);
```

#### Parameters

*This* 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\_HANDLE** 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 media 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 media the volume must be re-opened with **OpenVolume()**. If the new media 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 filesystem type.
EFI_NO_MEDIA	The device has no media.
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\_HANDLE Protocol

### Summary

Provides file based access to supported file systems.

### Revision Number

```
#define EFI_FILE_HANDLE_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, *EFI_FILE_HANDLE;
```

### Parameters

<i>Revision</i>	The version of the <b>EFI_FILE_HANDLE</b> interface. The version specified by this specification is 0x00010000. Future versions are required to be backward compatible to version 1.0.
<i>Open</i>	Opens or creates a new file. See Section 10.2.1.
<i>Close</i>	Closes the current file handle. See Section 0.
<i>Delete</i>	Deletes a file. See Section 10.2.3.
<i>Read</i>	Reads bytes from a file. See Section 10.2.4.
<i>Write</i>	Writes bytes to a file. See Section 10.2.5.
<i>GetPosition</i>	Returns the current file position. See Section 10.2.7.
<i>SetPosition</i>	Sets the current file position. See Section 10.2.6.
<i>GetInfo</i>	Gets the requested file or volume information. See Section 10.2.8.
<i>SetInfo</i>	Sets the requested file information. See Section 10.2.9.
<i>Flush</i>	Flushes all modified data associated with the file to the device. See Section 10.2.10.

## Description

The **EFI\_FILE\_HANDLE** protocol provides access to supported file systems . While it is likely that all files on a volume have the same **EFI\_FILE\_IO\_INTERFACE**, the specific interface used for any file handle is the one provided with the file handle.

An **EFI\_FILE\_HANDLE** 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.

A file system driver may cache data relating to an open file. The **Flush()** function is provided that flushes all dirty data in the file system, relative to the requested file, to the physical media. 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_HANDLE    This,
    OUT EFI_FILE_HANDLE   *NewHandle,
    IN CHAR16             *FileName,
    IN UINT64             OpenMode,
    IN UINT64             Attributes
);
```

### Parameters

<i>This</i>	The file handle to the source location. This would typically be an open handle to a directory. Type <b>EFI_FILE_HANDLE</b> is defined in Section 10.2.
<i>NewHandle</i>	A pointer to the location to return the opened handle for the new file. Type <b>EFI_FILE_HANDLE</b> is defined in Section 10.2.
<i>FileName</i>	The Null-terminated string of the name of the file to be opened. The file name may contain the following path modifiers: “\”, “.”, and “..”.
<i>OpenMode</i>	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”.
<i>Attributes</i>	Only valid for <b>EFI_FILE_MODE_CREATE</b> , 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 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* resides 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 media of the device changes all accesses, including the File handle, will result in **EFI\_MEDIA\_CHANGED**. To access the new media, 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 media.
EFI_MEDIA_CHANGED	The device has a different media in it or the media is no longer supported.
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 was not opened.
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_HANDLE    This
);
```

#### Parameters

*This*                      The file handle to close. Type **EFI\_FILE\_HANDLE** 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_HANDLE    This  
);
```

#### Parameters

*This* The handle to the file to delete. Type **EFI\_FILE\_HANDLE** 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
(EFI_API *EFI_FILE_READ) (
    IN EFI_FILE_HANDLE    This,
    IN OUT UINTN          *BufferSize,
    OUT VOID              *Buffer
);
```

### Parameters

<i>This</i>	The file handle to read data from. Type <b>EFI_FILE_HANDLE</b> is defined in Section 10.2.
<i>BufferSize</i>	On input, the size of the <i>Buffer</i> . On output, the amount of data returned in <i>Buffer</i> . In both cases, the size is measured in bytes.
<i>Buffer</i>	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 media.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_BUFFER_TOO_SMALL	The <i>BufferSize</i> is too small to read the current directory entry. <i>BufferSize</i> 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_HANDLE    This,
    IN OUT UINTN          *BufferSize,
    IN VOID               *Buffer
);
```

### Parameters

<i>This</i>	The file handle to write data to. Type <b>EFI_FILE_HANDLE</b> is defined in Section 10.2.
<i>BufferSize</i>	On input, the size of the <i>Buffer</i> . On output, the amount of data actually written. In both cases, the size is measured in bytes.
<i>Buffer</i>	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_UNSUPPORTED	Writes to open directory files are not supported.
EFI_NO_MEDIA	The device has no media.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	The file or media 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_HANDLE    This,
    IN UINT64              Position
);
```

### Parameters

*This* The file to set the requested position on. Type **EFI\_FILE\_HANDLE** 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
(EFI_API *EFI_GET_POSITION) (
    IN EFI_FILE_HANDLE    This,
    OUT UINT64             *Position
);
```

### Parameters

*This* The file to get the current position on. Type **EFI\_FILE\_HANDLE** 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_HANDLE    This,
    IN EFI_GUID            *InformationType,
    IN OUT UINTN           *BufferSize,
    OUT VOID               *Buffer
);
```

### Parameters

<i>This</i>	The file the requested information is for. Type <b>EFI_FILE_HANDLE</b> is defined in Section 10.2.
<i>InformationType</i>	The type identifier for the information being requested. Type <b>EFI_GUID</b> is defined in Chapter 3.
<i>BufferSize</i>	On input, the size of <i>Buffer</i> . On output, the amount of data returned in <i>Buffer</i> . In both cases, the size is measured in bytes.
<i>Buffer</i>	A pointer to the data buffer to return. The buffer's type is indicated by <i>InformationType</i> .

### 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 <i>InformationType</i> is not known.
EFI_NO_MEDIA	The device has no media.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	The file or media is write protected.
EFI_ACCESS_DENIED	The file was opened read only.
EFI_BUFFER_TOO_SMALL	The <i>BufferSize</i> is too small to read the current directory entry. <i>BufferSize</i> 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_HANDLE    This,
    IN EFI_GUID            *InformationType,
    IN UINTN               BufferSize,
    OUT VOID               *Buffer
);
```

### Parameters

<i>This</i>	The file the information is for. Type <b>EFI_FILE_HANDLE</b> is defined in Section 10.2.
<i>InformationType</i>	The type identifier for the information being set. Type <b>EFI_GUID</b> is defined in Chapter 3.
<i>BufferSize</i>	The size, in bytes, of <i>Buffer</i> .
<i>Buffer</i>	A pointer to the data buffer to write. The buffer's type is indicated by <i>InformationType</i> .

### 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 <i>InformationType</i> is not known.
EFI_NO_MEDIA	The device has no media.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	The file or media is write protected.
EFI_ACCESS_DENIED	The file was opened read-only.
EFI_VOLUME_FULL	The volume is full.
EFI_BUFFER_TOO_SMALL	The <i>BufferSize</i> is too small to read the current directory entry. <i>BufferSize</i> has been updated with the size needed to complete the request.

## 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_HANDLE    This
);
```

### Parameters

*This* The file to flush. Type **EFI\_FILE\_HANDLE** 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 media.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	The file or media is write protected.
EFI_ACCESS_DENIED	The file was opened read-only.
EFI_VOLUME_FULL	The volume is full.

## 10.2.11 EFI\_GENERIC\_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 \
    { 9576e92-6d3f-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_RESERVIED       0x0000000000000008
#define EFI_FILE_DIRECTORY      0x0000000000000010
#define EFI_FILE_ARCHIVE        0x0000000000000020
#define EFI_FILE_VALID_ATTR     0x0000000000000037
```

### Parameters

<i>Size</i>	Size of the <b>EFI_FILE_INFO</b> structure, including the Null-terminated Unicode <i>FileName</i> string.
<i>FileSize</i>	The size of the file in bytes.
<i>PhysicalSize</i>	The amount of physical space the file consumes on the file system volume.
<i>CreateTime</i>	The time the file was created.
<i>LastAccessTime</i>	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\_GENERIC\_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.
- 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.

### GUID

```
#define EFI_FILE_SYSTEM_INFO_ID \
    { 9576e93-6d3f-11d2-8e39-00a0c969723b }
```

### Related Definitions

```
typedef struct {
    UINT64                Size;
    BOOLEAN               ReadOnly;
    UINT64                VolumeSize;
    UINT64                FreeSpace;
    UINT32                BlockSize;
    CHAR16                VolumeLabel[];
} EFI_FILE_SYSTEM_INFO;
```

### Parameters

<i>Size</i>	Size of the <b>EFI_FILE_SYSTEM_INFO</b> structure, including the Null-terminated Unicode <i>VolumeLabel</i> string.
<i>ReadOnly</i>	<b>TRUE</b> if the volume only supports read-only access.
<i>VolumeSize</i>	The number of bytes managed by the file system.
<i>FreeSpace</i>	The number of bytes available for use by the file system.
<i>BlockSize</i>	The nominal block size files are typically grown by.
<i>VolumeLabel</i>	The Null-terminated string that is the volume's label.

### Description

The **EFI\_FILE\_SYSTEM\_INFO** data structure is a read-only 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.

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
);
```

#### Parameters

<i>This</i>	Indicates the calling context. Type <b>EFI_LOAD_FILE_INTERFACE</b> is defined in Section 11.1.
<i>FilePath</i>	The device specific path of the file to load. Type <b>EFI_DEVICE_PATH</b> is defined in Chapter 3.
<i>BootPolicy</i>	If <b>TRUE</b> , indicates that the request originates from the boot manager, and that the boot manager is attempting to load <i>FilePath</i> as a boot selection.
<i>BufferSize</i>	On input the size of <i>Buffer</i> in bytes. On output with a return code of <b>EFI_SUCCESS</b> , the amount of data transferred to <i>Buffer</i> . On output with a return code of <b>EFI_BUFFER_TOO_SMALL</b> , the size of <i>Buffer</i> required to retrieve the requested file.
<i>Buffer</i>	The memory buffer to transfer the file to.

#### 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 *BufferSize* is not large enough to hold the entire file, the error **EFI\_BUFFER\_TOO\_SMALL** is returned, *BufferSize* is updated to indicate the size of the buffer needed to obtain the file, and 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, the error **EFI\_NOT\_FOUND** 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 boot selection where the device path points to a removable media device such as DVD, and the *FilePath* merely points to the root of the device, the firmware interprets this as wanting to boot from the first valid loader. The firmware enumerates the files in the boot catalog of the DVD looking for the first EFI loader for the local platform and then loads that image. So in this case, the interpretation of the *FilePath* is device-specific and may, based on the appropriate policy for the device, return a file that is not literally specified by *FilePath*. However, the returned file would be the proper EFI Loader image for the inexact *FilePath* as defined by the device's boot policy.

### Status Codes Returned

EFI_SUCCESS	The file was loaded.
EFI_NO_SUCH_MEDIA	No media 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.



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

<i>Revision</i>	The revision to which the <b>SERIAL_IO_INTERFACE</b> adheres. All future revisions must be backwards compatible. If a future version is not backwards compatible, it is not the same GUID.
<i>Reset</i>	Resets the hardware device.
<i>SetAttributes</i>	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.

<i>SetControl</i>	Set the control bits on a serial device. These include Request to Send and Data Terminal Ready.
<i>GetControl</i>	Read the status of the control bits on a serial device. These include Clear to Send, Data Set Ready, Ring Indicator, and Carrier Detect.
<i>Write</i>	Send a buffer of characters to a serial device.
<i>Read</i>	Receive a buffer of characters from a serial device.
<i>Mode</i>	Pointer to <b>SERIAL_IO_MODE</b> data. Type <b>SERIAL_IO_MODE</b> is defined in “Related Definitions”.

## Related Definitions

```

//*****
// SERIAL_IO_MODE
//*****
typedef struct {
    UINT32                                     ControlMask;

    // current Attributes
    UINT32                                     Timeout;
    UINT64                                     BaudRate;
    UINT32                                     ReceiveFifoDepth;
    UINTN                                     DataBits;
    EFI_PARITY_TYPE                           Parity;
    EFI_STOP_BITS_TYPE                         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:

<i>ControlMask</i>	A mask of the Control bits that the device supports. The device must always support the Input Buffer Empty control bit.
<i>Timeout</i>	If applicable, the number of microseconds to wait before timing out a Read or Write operation.
<i>BaudRate</i>	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 devices designed speed.
<i>ReceiveFifoDepth</i>	The number of characters the device will buffer on input.
<i>DataBits</i>	The number of data bits in each character.

*Parity* If applicable, this is the type of parity 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 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 UINTN                  BaudRate,
    IN UINTN                  ReceiveFifoDepth,
    IN UINTN                  Timeout
    IN EFI_PARITY_TYPE        Parity,
    IN UINT8                  DataBits,
    IN EFI_STOP_BITS_TYPE     StopBits,
);
```

### Parameters

<i>This</i>	A pointer to the <b>SERIAL_IO_INTERFACE</b> instance. Type <b>SERIAL_IO_INTERFACE</b> is defined in Section 12.1.
<i>BaudRate</i>	The requested baud rate. A <i>BaudRate</i> value of 0 will use the device's default interface speed.
<i>ReceiveFifoDepth</i>	The requested depth of the FIFO on the receive side of the serial interface. A <i>ReceiveFifoDepth</i> value of 0 will use the device's default FIFO depth.
<i>Timeout</i>	The requested time out for a single character in microseconds. This timeout applies to both the transmit and receive side of the interface. A <i>Timeout</i> value of 0 will use the device's default time out value.
<i>Parity</i>	The type of parity to use on this serial device. A <i>Parity</i> value of <b>DefaultParity</b> will use the device's default parity value. Type <b>EFI_PARITY_TYPE</b> is defined in Section 12.1.
<i>DataBits</i>	The number of data bits to use on this serial device. A <i>DataBits</i> value of 0 will use the device's default data bit setting.
<i>StopBits</i>	The number of stop bits to use on this serial device. A <i>StopBits</i> value of <b>DefaultStopBits</b> will use the device's default number of stop bits. Type <b>EFI_STOP_BITS_TYPE</b> 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, and the selected baud rate will be returned in *BaudRate*. 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
(EFI_API *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
```

#### Description

The **SetControl()** function is used to assert or deassert the control signals on a serial device. The following signals are set according to their bit settings:

Request to Send

Data Terminal Ready

#### 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

<i>This</i>	A pointer to the <b>SERIAL_IO_INTERFACE</b> instance. Type <b>SERIAL_IO_INTERFACE</b> is defined in Section 12.1.
<i>Control</i>	A pointer to return the current Control signals from the serial device.

### 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

<i>This</i>	A pointer to the <b>SERIAL_IO_INTERFACE</b> instance. Type <b>SERIAL_IO_INTERFACE</b> is defined in Section 12.1.
<i>BufferSize</i>	On input, the size of the <i>Buffer</i> . On output, the amount of data returned in <i>Buffer</i> .
<i>Buffer</i>	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.



# 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

<i>StriColl</i>	Performs a case-insensitive comparison of two Null-terminated Unicode strings. See Section 13.1.1.
<i>MetaiMatch</i>	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.
<i>StrLwr</i>	Converts all the Unicode characters in a Null-terminated Unicode string to lower case Unicode characters. See Section 0.

<i>StrUpr</i>	Converts all the Unicode characters in a Null-terminated Unicode string to upper case Unicode characters. See Section 13.1.4 .
<i>FatToStr</i>	Converts an 8.3 FAT file name using an OEM character set to a Null-terminated Unicode string. See Section 13.1.5.
<i>StrToFat</i>	Converts a Null-terminated Unicode string to legal characters in a FAT filename using an OEM character set. See Section 13.1.6.
<i>SupportedLanguages</i>	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  
(EFI_API *EFI_UNICODE_COLLATION_STRICOLL) (  
    IN UNICODE_COLLATION_INTERFACE  *This,  
    IN CHAR16                        *s1,  
    IN CHAR16                        *s2  
);
```

#### Parameters

<i>This</i>	A pointer to the <b>UNICODE_COLLATION</b> instance. Type <b>UNICODE_COLLATION_INTERFACE</b> is defined in Section 13.1.
<i>s1</i>	A pointer to a Null-terminated Unicode string.
<i>s2</i>	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

<i>This</i>	A pointer to the <b>UNICODE_COLLATION_INTERFACE</b> instance. Type <b>UNICODE_COLLATION_INTERFACE</b> is defined in Section 13.1.
<i>String</i>	A pointer to a Null-terminated Unicode string.
<i>Pattern</i>	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*.

<i>*</i>	Match 0 or more characters.
<i>?</i>	Match any one character.
<i>[ &lt;char1&gt;&lt;char2&gt;...&lt;charN&gt; ]</i>	Match any character in the set.
<i>[ &lt;char1&gt;-&lt;char2&gt; ]</i>	Match any character between <char1> and <char2>.
<i>&lt;char&gt;</i>	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

<i>This</i>	A pointer to the <b>UNICODE_COLLATION_INTERFACE</b> instance. Type <b>UNICODE_COLLATION_INTERFACE</b> is defined in Section 13.1.
<i>String</i>	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

<i>This</i>	A pointer to the <b>UNICODE_COLLATION_INTERFACE</b> instance. Type <b>UNICODE_COLLATION_INTERFACE</b> is defined in Section 13.1.
<i>String</i>	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

<i>This</i>	A pointer to the <b>UNICODE_COLLATION_INTERFACE</b> instance. Type <b>UNICODE_COLLATION_INTERFACE</b> is defined in Section 13.1.
<i>FatSize</i>	The size of the string <i>Fat</i> in bytes.
<i>Fat</i>	A pointer to a Null-terminated string that contains an 8.3 file name using an OEM character set.
<i>String</i>	A pointer to a Null-terminated Unicode string.

#### 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

<i>This</i>	A pointer to the <b>UNICODE_COLLATION_INTERFACE</b> instance. Type <b>UNICODE_COLLATION_INTERFACE</b> is defined in Section 13.1.
<i>String</i>	A pointer to a Null-terminated Unicode string.
<i>FatSize</i>	The size of the string <i>Fat</i> in bytes.
<i>Fat</i>	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*. 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	<i>Fat</i> is an 8.3 file name.
FALSE	<i>Fat</i> is a Long File Name.





# PXE Base Code Protocol

This chapter defines the PXE Base Code protocol. This protocol is used to access PXE (Preboot Execution Environment)-compatible devices.

## 14.1 EFI\_PXE\_BASE\_CODE Protocol

### Summary

The **EFI\_PXE\_BASE\_CODE** protocol is used to control PXE-compatible devices. The features of PXE compatible device 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** protocol may be used by the firmware's boot manager to support remote boot functionality.

### 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
    EFI_SIMPLE_NETWORK
    EFI_PXE_BASE_CODE_START
    EFI_PXE_BASE_CODE_STOP
    EFI_PXE_BASE_CODE_DHCP
    EFI_PXE_BASE_CODE_DISCOVER
    EFI_PXE_BASE_CODE_MTFTP
    EFI_PXE_BASE_CODE_UDP_WRITE
    EFI_PXE_BASE_CODE_UDP_READ
    EFI_PXE_BASE_CODE_SET_IP_FILTER
    EFI_PXE_BASE_CODE_ARP
    EFI_PXE_BASE_CODE_SET_PARAMETERS
    EFI_PXE_BASE_CODE_SET_STATION_IP
    EFI_PXE_BASE_CODE_SET_PACKETS
    EFI_PXE_BASE_CODE_MODE
} EFI_PXE_BASE_CODE;

Revision;
*SimpleNetwork;
Start;
Stop;
Dhcp;
Discover;
Mtftp;
UdpWrite;
UdpRead;
SetIpFilter;
Arp;
SetParameters;
SetStationIp;
SetPackets;
*Mode;
```

## Parameters

<i>Revision</i>	The revision of the <b>EFI_PXE_BASE_CODE</b> Protocol.
<i>Start</i>	Used to start the PXE Base Code Protocol.
<i>Stop</i>	Used to stop the PXE Base Code Protocol.
<i>Dhcp</i>	Attempts to complete a DHCPv4 D.O.R.A. (discover / offer / request / acknowledge) or DHCPv6 S.A.R.R (solicit / advertise / request / reply) sequence.
<i>Discover</i>	Attempts to complete the PXE Boot Server and/or boot image discovery sequence.
<i>Mtftp</i>	Used to perform TFTP and MTFTP services.
<i>UdpWrite</i>	Writes a UDP packet to the network interface.
<i>UdpRead</i>	Reads a UDP packet from the network interface.
<i>SetIpFilter</i>	Updates the IP receive filters of the network device.
<i>Arp</i>	Uses the ARP protocol to resolve a MAC address.
<i>SetParameters</i>	Updates the parameters that affect the operation of the PXE Base Code Protocol.
<i>SetStationIp</i>	Updates the station IP address and subnet mask values.
<i>SetPackets</i>	Updates the contents of the cached DHCP and Discover packets.
<i>Mode</i>	Pointer to the <b>EFI_PXE_BASE_CODE_MODE</b> data for this device. The <b>EFI_PXE_BASE_CODE_MODE</b> structure is defined in "Related Definitions".

The following data values in the **EFI\_PXE\_BASE\_CODE\_MODE** structure are read-only and are updated by the code that produces the **EFI\_PXE\_BASE\_CODE** protocol functions:

<i>Started</i>	<b>TRUE</b> if this device has been started by calling <b>Start()</b> . This field is set to <b>TRUE</b> by the <b>Start()</b> function and to <b>FALSE</b> by the <b>Stop()</b> function.
<i>Ipv6Supported</i>	<b>TRUE</b> if this device supports IPv6.
<i>UsingIpv6</i>	<b>TRUE</b> if this device is currently using IPv6. This field is set by <b>Start()</b> function.
<i>BisSupported</i>	<b>TRUE</b> if this device supports Boot Integrity Services (BIS). This field is set by <b>Start()</b> function.
<i>BisDetected</i>	<b>TRUE</b> if this device and the platform support Boot Integrity Services (BIS). This field is set by <b>Start()</b> function.

<i>Callback</i>	Pointer to the callback function that is invoked by the PXE Base Code Protocol when it is waiting for an event. If this field is <b>NULL</b> , then no callbacks will be generated. This field is initialized to <b>NULL</b> by the <b>Start()</b> function, and can be modified with the <b>SetParameters()</b> function.
<i>AutoArp</i>	<b>TRUE</b> for automatic ARP packet generation. <b>FALSE</b> for no automatic ARP packet generation. This field is initialized to <b>TRUE</b> by <b>Start()</b> , and can be modified with the <b>SetParameters()</b> function.
<i>StationIp</i>	The device's current IP address. This field is initialized to a zero address by <b>Start()</b> . This field is also set by the <b>Dhcp()</b> function, and can be modified with the <b>SetStationIp()</b> function.
<i>SubnetMask</i>	The device's current subnet mask. This field is initialized to a zero address by the <b>Start()</b> function. This field is also set by the <b>Dhcp()</b> function, and can be modified with the <b>SetStationIp()</b> function.
<i>DhcpDiscover</i>	Pointer to the cached DHCP Discover packet. This field is initialized to <b>NULL</b> by the <b>Start()</b> function, and is set by the <b>Dhcp()</b> and <b>SetPackets()</b> functions.
<i>DhcpAck</i>	Pointer to the cached DHCP Ack packet. This field is initialized to <b>NULL</b> by the <b>Start()</b> function, and is set by the <b>Dhcp()</b> and <b>SetPackets()</b> functions.
<i>ProxyOffer</i>	Pointer to the cached Proxy Offer packet. This field is initialized to <b>NULL</b> by the <b>Start()</b> function, and is set by the <b>Dhcp()</b> and <b>SetPackets()</b> functions.
<i>PxeDiscover</i>	Pointer to the cached PXE Discover packet. This field is initialized to <b>NULL</b> by the <b>Start()</b> function, and is set by the <b>Discover()</b> and <b>SetPackets()</b> functions.
<i>PxeReply</i>	Pointer to the cached PXE Reply packet. This field is initialized to <b>NULL</b> by the <b>Start()</b> function, and is set by the <b>Discover()</b> and <b>SetPackets()</b> functions.
<i>PxeBisReply</i>	Pointer to the cached PXE BIS Reply packet. This field is initialized to <b>NULL</b> by the <b>Start()</b> function, and is set by the <b>Discover()</b> and <b>SetPackets()</b> functions.
<i>IpFilter</i>	The current IP receive filter settings. The filter is initialized to disabled and 0 IP receive filters by the <b>Start()</b> function, and is set by the <b>SetIpFilter()</b> function.

<i>ArpCacheEntries</i>	The number of valid entries in the ARP cache. This field is reset to zero by the <b>Start()</b> function.
<i>ArpCache</i>	Pointer to an array of cached ARP entries.
<i>RouteTableEntries</i>	The number of valid entries in the current route table. This field is reset to zero by the <b>Start()</b> function.
<i>RouteTable</i>	Pointer to an array of route table entries.
<i>IcmpError</i>	A pointer to the most recent ICMP error packet. <b>NULL</b> if the previous operation did not generate an ICMP error. This field is initialized to <b>NULL</b> by the <b>Start()</b> function.
<i>TftpError</i>	A pointer to the most recent TFTP error packet. <b>NULL</b> if the previous operation did not generate a TFTP error. This field is initialized to <b>NULL</b> by the <b>Start()</b> function.

## Description

The basic mechanisms and flow for remote booting in EFI are identical to the remote boot functionality described in detail in the *Wired for Management Baseline Specification*. However, the actual execution environment, linkage, and calling conventions are replaced and enhanced for the EFI environment.

The DHCP Option for the Client Network Interface Identifier is used to inform the DHCP server that the client supports 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 IA-64 EFI environment. The server may use this information to provide default images in the case where 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.

## Related Definitions

```
typedef struct {
    BOOLEAN                Started;
    BOOLEAN                Ipv6Supported;
    BOOLEAN                UsingIpv6;
    BOOLEAN                BisSupported;
    BOOLEAN                BisDetected;
    EFI_PXE_BASE_CODE_CALLBACK CallBack;
    BOOLEAN                AutoArp;
    EFI_PXE_BASE_CODE_IP_ADDR StationIp;
    EFI_PXE_BASE_CODE_IP_ADDR 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;
    UINTN                  ArpCacheEntries;
    EFI_PXE_BASE_CODE_ARP_ENTRY *ArpCache;
    UINTN                  RouteTableEntries;
    EFI_PXE_BASE_CODE_ROUTE_ENTRY *RouteTable;
    EFI_PXE_BASE_CODE_ICMP_ERROR *IcmpError;
    EFI_PXE_BASE_CODE_TFTP_ERROR *TftpError;
} EFI_PXE_BASE_CODE_MODE;
```

The following section defines the data types for MAC addresses, IP addresses, and UDP ports. All of these data types use big endian byte ordering for their contents.

```
typedef UINT8 EFI_PXE_BASE_CODE_MAC_ADDR[16];

typedef UINT32 EFI_PXE_BASE_CODE_IPV4;

typedef UINT32 EFI_PXE_BASE_CODE_IPV6[4];

typedef union {
    EFI_PXE_BASE_CODE_IPV4_ADDR    Ipv4;
    EFI_PXE_BASE_CODE_IPV6_ADDR    Ipv6;
} EFI_PXE_BASE_CODE_IP_ADDR;

typedef UINT16 EFI_PXE_BASE_CODE_UDP_PORT;
```

The following section defines the data types for DHCP packets, ICMP error packets, and TFTP error packets. All of these are packed data structures.

```
typedef struct {
    UINT8                               BootpOpcode;
    UINT8                               BootpHwType;
    UINT8                               BootpHwAddrLen;
    UINT8                               BootpGateHops;
    UINT32                              BootpIdent;
    UINT16                              BootpSeconds;
    UINT16                              BootpFlags;
    EFI_PXE_BASE_CODE_IPV4              BootpCiAddr;
    EFI_PXE_BASE_CODE_IPV4              BootpYiAddr;
    EFI_PXE_BASE_CODE_IPV4              BootpSiAddr;
    EFI_PXE_BASE_CODE_IPV4              BootpGiAddr;
    EFI_PXE_BASE_CODE_MAC_ADDR          BootpHwAddr;
    UINT8                               BootpSrvName[64];
    UINT8                               BootpBootFile[128];
    UINT32                              DhcpMagik;
    UINT8                               DhcpOptions[56];
} EFI_PXE_BASE_CODE_DHCPV4_PACKET;

typedef struct {
} EFI_PXE_BASE_CODE_DHCPV6_PACKET;

typedef union {
    UINT8                               Raw[1472];
    EFI_PXE_BASE_CODE_DHCPV4_PACKET    Dhcpv4;
    EFI_PXE_BASE_CODE_DHCPV6_PACKET    Dhcpv6;
} EFI_PXE_BASE_CODE_PACKET;

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;
```

```
typedef struct {
    UINT8                                     ErrorCode;
    UINT8                                     ErrorString[127];
} EFI_PXE_BASE_CODE_TFTP_ERROR;
```

The following section defines the data types for IP receive filter settings.

```
typedef struct {
    UINTN                                     Filters;
    UINTN                                     IpCnt;
    EFI_PXE_BASE_CODE_IP_ADDR               *IpList;
} 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
```

The following section defines the data types for ARP cache entries, and route table entries.

```
typedef struct {
    EFI_PXE_BASE_CODE_IP_ADDR               IpAddr;
    EFI_PXE_BASE_CODE_MAC_ADDR              MacAddr;
} EFI_PXE_BASE_CODE_ARP_ENTRY;

typedef struct {
    EFI_PXE_BASE_CODE_IP_ADDR               IpAddr;
    EFI_PXE_BASE_CODE_IP_ADDR               SubnetMask;
    EFI_PXE_BASE_CODE_IP_ADDR               GwAddr;
} EFI_PXE_BASE_CODE_ROUTE_ENTRY;
```

The following 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
```

### 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

<i>This</i>	Pointer to the <b>EFI_PXE_BASE_CODE</b> instance.
<i>UseIpv6</i>	Specifies the type of IP addresses that are to be used during the session that is being started. Set to <b>TRUE</b> for IPv6 addresses, and <b>FALSE</b> 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\_INVALID\_PARAMETER** 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:

<i>Started</i>	Set to <b>TRUE</b> .
<i>Ipv6Supported</i>	No changes.
<i>UsingIpv6</i>	Set to <i>UseIpv6</i> .
<i>BisSupported</i>	No changes.
<i>BisDetected</i>	No changes.
<i>Callback</i>	Set to <b>NULL</b> .
<i>AutoArp</i>	Set to <b>TRUE</b> .
<i>StationIp</i>	Set to an address of all zeros.
<i>SubnetMask</i>	Set to a subnet mask of all zeros.



<i>DhcpDiscover</i>	Set to <b>NULL</b> .
<i>DhcpAck</i>	Set to <b>NULL</b> .
<i>ProxyOffer</i>	Set to <b>NULL</b> .
<i>PxeDiscover</i>	Set to <b>NULL</b> .
<i>PxeReply</i>	Set to <b>NULL</b> .
<i>PxeBisReply</i>	Set to <b>NULL</b> .
<i>IpFilter</i>	Set the <i>Filters</i> field to 0 and the <i>IpCnt</i> field to 0.
<i>ArpCacheEntries</i>	Set to 0.
<i>ArpCache</i>	No changes.
<i>RouteTableEntries</i>	Set to 0.
<i>RouteTable</i>	No changes.
<i>IcmpError</i>	Set to <b>NULL</b> .
<i>TftpError</i>	Set to <b>NULL</b> .

### Status Codes Returned

EFI_SUCCESS	The PXE Base Code Protocol was started.
EFI_INVALID_PARAMETER	One of the parameters is not valid.
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
(EFI_API *EFI_PXE_BASE_CODE_DHCP) (
    IN EFI_PXE_BASE_CODE    *This,
    IN BOOLEAN               SortOffers
);
```

#### Parameters

<i>This</i>	Pointer to the <b>EFI_PXE_BASE_CODE</b> instance.
<i>SortOffers</i>	<b>TRUE</b> if the offers received should be sorted. Set to <b>FALSE</b> 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 *StationIp*, *SubnetMask*, *DhcpDiscover*, *DhcpAck*, and *ProxyOffer* fields of the **EFI\_PXE\_BASE\_CODE\_MODE** structure are filled in. If a Proxy Offer packet is not received, then the *ProxyOffer* field is set to **NULL**. If the callback function that is invoked during the DHCP protocol packet transactions does not return **EFI\_PXE\_BASE\_CODE\_CALLBACK\_STATUS\_CONTINUE**, then **EFI\_ABORTED** will be returned. This function may take at least 31 seconds to return. If the DHCP Protocol does not complete, then **EFI\_TIMEOUT** will be returned. If *SortOffers* is **TRUE**, then the cached DHCP offers will be sorted before they are tried. If *SortOffers* is **FALSE**, then the cached DHCP offers will be tried in the order they are received. Please see the *Preboot Execution Environment (PXE) Specification* for additional details on the implementation of the DHCP Protocol.

## 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.

## 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 OUT EFI_PXE_BASE_CODE_DISCOVER_INFO *Info OPTIONAL
);
```

### Parameters

<i>This</i>	Pointer to the <b>EFI_PXE_BASE_CODE</b> instance.
<i>Type</i>	The type of bootstrap to perform.
<i>Layer</i>	Pointer to the boot server layer number to discover, which must be <b>PXE_BOOT_LAYER_INITIAL</b> 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 <i>Layer</i> , then the new <i>Layer</i> will be returned.
<i>UseBis</i>	<b>TRUE</b> if Boot Integrity Services are to be used.
<i>Info</i>	Pointer to a data structure that contains additional information on the type of discovery operation that is to be performed. If this field is <b>NULL</b> , then the contents of the cached <i>DhcpAck</i> and <i>ProxyOffer</i> packets will be used.

### 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 *PxeDiscover*, and *PxeReply* fields of the **EFI\_PXE\_BASE\_CODE\_MODE** structure are filled in. If *UseBis* is **TRUE**, then the *PxeBisReply* field of the **EFI\_PXE\_BASE\_CODE\_MODE** structure will also be filled in. If *UseBis* is **FALSE**, then *PxeBisReply* will be set to **NULL**. If the callback function that is invoked during the Discovery sequence does not return **EFI\_PXE\_BASE\_CODE\_CALLBACK\_STATUS\_CONTINUE**, then **EFI\_ABORTED** will be returned. This function may take at least 10 seconds to return. 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.

## Related Definitions

```
#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
//
// 17 through 32767 are reserved
// 32768 through 65279 are for vendor use
// 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

typedef struct {
    BOOLEAN                               UseMCast;
    BOOLEAN                               UseBCast;
    BOOLEAN                               UseUCast;
    BOOLEAN                               MustUseList;
    EFI_PXE_BASE_CODE_IP_ADDR             ServerMCastIp;
    UINTN                                  SrvListCnt;
    EFI_PXE_BASE_CODE_SRVLIST             *SrvList;
} EFI_PXE_BASE_CODE_DISCOVER_INFO;

typedef struct {
    UINT16                                 Type;
    UINTN                                  IpCnt;
    EFI_PXE_BASE_CODE_IP_ADDR             *IpAddr;
} EFI_PXE_BASE_CODE_SRVLIST;
```

## 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,
    IN BOOLEAN                    Overwrite,
    IN OUT UINTN                  *BufferSize,
    IN UINTN                      *BlockSize,    OPTIONAL
    IN EFI_PXE_BASE_CODE_IP_ADDR *ServerIp,
    IN UINT8                      *Filename,
    IN EFI_PXE_BASE_CODE_MTFTP_INFO *Info        OPTIONAL
);
```

#### Parameters

<i>This</i>	Pointer to the <b>EFI_PXE_BASE_CODE</b> instance.
<i>Operation</i>	The type of operation to perform. See "Related Definitions" for the list of operation types.
<i>BufferPtr</i>	A pointer to the data buffer.
<i>Overwrite</i>	Only used on write file operations. <b>TRUE</b> if a file on a remote server can be overwritten.
<i>BufferSize</i>	For read file and write file operations, this is the size of the buffer specified by <i>BufferPtr</i> . For read file operations, if <i>BufferSize</i> is smaller than the size of the file being read, then this field will return the required size. This field must be at least 128 bytes for TFTP read directory operations, and at least 128 bytes plus the size of the <b>EFI_PXE_BASE_CODE_IP_ADDR</b> structure for MTFTP read directory operations. For get file size operations, this field returns the size of the requested file.
<i>BlockSize</i>	The requested block size to be used during a TFTP transfer. This must be at least 512. If this field is <b>NULL</b> , then the largest block size supported by the implementation will be used.
<i>ServerIp</i>	The TFTP / MTFTP server IP address.



<i>Filename</i>	A Null-terminated ASCII string that specifies a directory name or a file name.
<i>Info</i>	Pointer to the MTFTP information. This information is required to start or join a multicast TFTP session. See "Related Definitions" for the description of this data structure.

## 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 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 get file size operations, the size of the requested file is returned in *BufferSize*, and **EFI\_SUCCESS** will be returned if the TFTP server supports this option.

## Related Definitions

```
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_READ_FILE,
    EFI_PXE_BASE_CODE_MTFTP_READ_DIRECTORY,
    EFI_PXE_BASE_CODE_MTFTP_LAST
} EFI_PXE_BASE_CODE_TFTP_OPCODE;

typedef struct {
    EFI_PXE_BASE_CODE_IP_ADDR      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;
```

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 EFI_PXE_BASE_CODE_OPFLAGS  OpFlags,
    IN EFI_PXE_BASE_CODE_IP_ADDR  *DestIp,
    IN EFI_PXE_BASE_CODE_UDP_PORT *DestPort,
    IN EFI_PXE_BASE_CODE_IP_ADDR  *GatewayIp,  OPTIONAL
    IN EFI_PXE_BASE_CODE_IP_ADDR  *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

<i>This</i>	Pointer to the <b>EFI_PXE_BASE_CODE</b> instance.
<i>OpFlags</i>	The UDP operation flags. If <b>MAY_FRAGMENT</b> is set, then if required, this UDP write operation may be broken up across multiple packets.
<i>DestIp</i>	The destination IP address.
<i>DestPort</i>	The destination UDP port number.
<i>GatewayIp</i>	The gateway IP address. If <i>DestIp</i> is not in the same subnet as <i>StationIp</i> , then this gateway IP address will be used. If this field is <b>NULL</b> , and the <i>DestIp</i> is not in the same subnet as <i>StationIp</i> , then the <i>RouteTable</i> will be used.
<i>SrcIp</i>	The source IP address. If this field is <b>NULL</b> , then <i>StationIp</i> will be used as the source IP address.
<i>SrcPort</i>	The source UDP port number. If <i>OpFlags</i> has <b>ANY_SRC_PORT</b> set or <i>SrcPort</i> is <b>NULL</b> , then a source UDP port will be automatically selected. If a source UDP port was automatically selected, and <i>SrcPort</i> is not <b>NULL</b> , then it will be returned in <i>SrcPort</i> .
<i>HeaderSize</i>	An optional field which may be set to the length of a header at <i>HeaderPtr</i> to be prepended to the data at <i>BufferPtr</i> .

<i>HeaderPtr</i>	If <i>HeaderSize</i> is not <b>NULL</b> , a pointer to a header to be prepended to the data at <i>BufferPtr</i> .
<i>BufferSize</i>	A pointer to the size of the data at <i>BufferPtr</i> .
<i>BufferPtr</i>	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 **EFI\_ICMP\_ERROR** will be returned. If the callback function that is invoked during the UDP write operation 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 EFI_PXE_BASE_CODE_OPFLAGS  OpFlags,
    IN OUT EFI_PXE_BASE_CODE_IP_ADDR *DestIp,      OPTIONAL
    IN OUT EFI_PXE_BASE_CODE_UDP_PORT *DestPort,   OPTIONAL
    IN OUT EFI_PXE_BASE_CODE_IP_ADDR *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

<i>This</i>	Pointer to the <b>EFI_PXE_BASE_CODE</b> instance.
<i>OpFlags</i>	The UDP operation flags.
<i>DestIp</i>	The destination IP address.
<i>DestPort</i>	The destination UDP port number.
<i>SrcIp</i>	The source IP address.
<i>SrcPort</i>	The source UDP port number.
<i>HeaderSize</i>	An optional field which may be set to the length of a header to be put in <i>HeaderPtr</i> .
<i>HeaderPtr</i>	If <i>HeaderSize</i> is not <b>NULL</b> , a pointer to a buffer to hold the <i>HeaderSize</i> bytes which follow the IP header.
<i>BufferSize</i>	On input, a pointer to the size of the buffer at <i>BufferPtr</i> . On output, the size of the data written to <i>BufferPtr</i> .
<i>BufferPtr</i>	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. 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.

### Destination IP Filter Operation

OpFlags USE_FILTER	OpFlags ANY_DEST_IP	DestIp	Action
0	0	NULL	Receive a packet sent to <i>StationIp</i> .
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 <i>DestIp</i> .
0	1	not NULL	Receive a packet sent to any IP address and, return the destination IP address in <i>DestIp</i> .
1	x	not NULL	Receive a packet whose destination IP address passes the IP filter, and return the destination IP address in <i>DestIp</i> .

### Destination UDP Port Filter Operation

OpFlags ANY_DEST_PORT	DestPort	Action
0	NULL	Return <b>EFI_INVALID_PARAMETER</b> .
1	NULL	Receive a packet sent to any UDP port.
0	not NULL	Receive a packet whose destination Port matches <i>DestPort</i> .
1	not NULL	Receive a packet sent to any UDP port, and return the destination port in <i>DestPort</i> .

## Source IP Filter Operation

OpFlags ANY_SRC_IP	SrcIp	Action
0	NULL	Return <b>EFI_INVALID_PARAMETER</b> .
1	NULL	Receive a packet sent from any IP address.
0	not NULL	Receive a packet whose source IP address matches <i>SrcIp</i> .
1	not NULL	Receive a packet sent from any IP address, and return the source IP address in <i>SrcIp</i> .

## Source UDP Port Filter Operation

OpFlags ANY_SRC_PORT	SrcPort	Action
0	NULL	Return <b>EFI_INVALID_PARAMETER</b> .
1	NULL	Receive a packet sent from any UDP port.
0	not NULL	Receive a packet whose source UDP port matches <i>SrcPort</i> .
1	not NULL	Receive a packet sent from any UDP port, and return the source UDP port in <i>SrcPort</i> .

## 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 <i>Buffer</i> 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*. If **Dhcp()**, **Discover()**, or **Mtftp()** is called, then the IP receive filter settings will be left in an undefined state. Depending on the *OpFlags* used in a **UdpRead()** or a **UdpWrite()** call, the IP receive filter settings may also be left in an undefined state. If an application or driver wishes to preserve the IP receive filter settings, they will have to preserve the IP receive filter settings prior to these calls, and use **SetIpFilter()** to restore them afterward. 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 IP's 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_PXE_BASE_CODE_IP_ADDR  *IpAddr,
    IN EFI_PXE_BASE_CODE_MAC_ADDR *MacAddr    OPTIONAL
);
```

#### Parameters

*This* Pointer to the **EFI\_PXE\_BASE\_CODE** instance.

*IpAddr* This is a 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 this is not **NULL**, then this is a pointer to the MAC address that was resolved with the ARP protocol.

#### Description

This function uses the ARP protocol to resolve 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 the *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** will be returned. If the callback function that is invoked during the ARP protocol packet transactions does not return **EFI\_PXE\_BASE\_CODE\_CALLBACK\_STATUS\_CONTINUE**, then **EFI\_ABORTED** will be 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 EFI_PXE_BASE_CODE_CALLBACK *NewCallBack  OPTIONAL
);
```

#### Parameters

<i>This</i>	Pointer to the <b>EFI_PXE_BASE_CODE</b> instance.
<i>NewAutoArp</i>	If not <b>NULL</b> , a pointer to a value to replace the current value of <i>AutoARP</i> ( <b>TRUE</b> for automatic ARP packet generation. <b>FALSE</b> for no automatic ARP packet generation). If <b>NULL</b> , this parameter is ignored.
<i>NewCallBack</i>	If not <b>NULL</b> , a pointer to a value to replace the current value of <i>CallBack</i> . That value may be a pointer to the callback function that is invoked by the PXE Base Code Protocol when it is waiting for an event, or <b>NULL</b> , in which case no callbacks will be generated.

#### 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 *NewCallBack* field is used to register a callback function that is called when the PXE Base Code Protocol is waiting for an event. If *NewCallBack* is **NULL**, then no callbacks will be generated. This function updates the *CallBack* field of the **EFI\_PXE\_BASE\_CODE\_MODE** structure to *NewCallBack*.

#### 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.

#### Prototype

```
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_SET_STATION_IP) (
    IN EFI_PXE_BASE_CODE          *This,
    IN EFI_PXE_BASE_CODE_IP_ADDR  *NewStationIp,    OPTIONAL
    IN EFI_PXE_BASE_CODE_IP_ADDR  *NewSubnetMask    OPTIONAL
);
```

#### Parameters

*This* Pointer to the **EFI\_PXE\_BASE\_CODE** instance.

*NewStationIp* The new IP address to be used by this network device. If this field is **NULL**, then the *StationIp* address will not be modified.

*NewSubnetMask* The new subnet mask to be used by this network device. If this field is **NULL**, then the *SubnetMask* will not be modified.

#### Description

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 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

<i>This</i>	Pointer to the <b>EFI_PXE_BASE_CODE</b> instance.
<i>NewDhcpDiscover</i>	Pointer to the new cached DHCP Discover packet.
<i>NewDhcpAck</i>	Pointer to the new cached DHCP Ack packet.
<i>NewProxyOffer</i>	Pointer to the new cached Proxy Offer packet.
<i>NewPxeDiscover</i>	Pointer to the new cached PXE Discover packet.
<i>NewPxeReply</i>	Pointer to the new cached PXE Reply packet.
<i>NewPxeBisReply</i>	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.

### 14.1.13 EFI\_PXE\_BASE\_CODE\_CALLBACK

#### Summary

Callback function that is invoked when the PXE Base Code Protocol is waiting for an event.

#### Prototype

```
EFI_PXE_BASE_CODE_CALLBACK_STATUS
(*EFI_PXE_BASE_CODE_CALLBACK) (
    IN EFI_PXE_BASE_CODE           *This,
    IN EFI_PXE_BASE_CODE_FUNCTION  Function,
    IN BOOLEAN                     Received,
    IN UINTN                       PacketLen,
    IN EFI_PXE_BASE_CODE_PACKET    *Packet    OPTIONAL
);
```

#### Parameters

<i>This</i>	Pointer to the <b>EFI_PXE_BASE_CODE</b> instance.
<i>Function</i>	The PXE Base Code Protocol function that is waiting for an event.
<i>Received</i>	<b>TRUE</b> if the callback is being invoked due to a receive event. <b>FALSE</b> if the callback is being invoked due to a transmit event.
<i>PacketLen</i>	The length, in bytes, of <i>Packet</i> . This field will be 0 if this is not a packet event.
<i>Packet</i>	If <i>Received</i> is <b>TRUE</b> , then this is a pointer to the packet that was just received. If <i>Received</i> is <b>FALSE</b> , then this is a pointer to the packet that is about to be transmitted. This field will be <b>NULL</b> if this is not a packet event.

#### Description

This function is invoked when the PXE Base Code Protocol is waiting for an event. The type of event is specified by *Function* and *Received*. *PacketLength* and *Packet* specify the packet that generated the event. If these fields are 0 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.

## Related Definitions

```
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;
```

```
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_MTFPT,  
    EFI_PXE_BASE_CODE_FUNCTION_UDP_WRITE,  
    EFI_PXE_BASE_CODE_FUNCTION_UDP_READ,  
    EFI_PXE_BASE_CODE_FUNCTION_ARP,  
    EFI_PXE_BASE_CODE_PXE_FUNCTION_LAST  
} EFI_PXE_BASE_CODE_FUNCTION;
```

# 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
    EFI_SIMPLE_NETWORK_START           Revision;
    EFI_SIMPLE_NETWORK_STOP           Start;
    EFI_SIMPLE_NETWORK_INITIALIZE     Stop;
    EFI_SIMPLE_NETWORK_RESET          Initialize;
    EFI_SIMPLE_NETWORK_SHUTDOWN       Reset;
    EFI_SIMPLE_NETWORK_RECEIVE_FILTERS Shutdown;
    EFI_SIMPLE_NETWORK_STATION_ADDRESS ReceiveFilters;
    EFI_SIMPLE_NETWORK_STATISTICS     StationAddress;
    EFI_SIMPLE_NETWORK_MCAST_IP_TO_MAC Statistics;
    EFI_SIMPLE_NETWORK_NVDATA         MCastIpToMac;
    EFI_SIMPLE_NETWORK_GET_STATUS     NvData;
    EFI_SIMPLE_NETWORK_TRANSMIT       GetStatus;
    EFI_SIMPLE_NETWORK_RECEIVE        Transmit;
    EFI_EVENT                         Receive;
    EFI_SIMPLE_NETWORK_MODE           WaitForPacket;
    } EFI_SIMPLE_NETWORK;
```

## Parameters

<i>Revision</i>	Revision of the <b>EFI_SIMPLE_NETWORK</b> Protocol.
<i>Start</i>	Used to prepare the network interface for further command operations. No other <b>EFI_SIMPLE_NETWORK</b> interface functions will operate until this call is made.
<i>Stop</i>	Used to stop any further network interface command processing. No other <b>EFI_SIMPLE_NETWORK</b> interface functions will operate after this call is made until another <i>Start</i> call is made.
<i>Initialize</i>	This function resets the network adapter and allocates the transmit and receive buffers.
<i>Reset</i>	This function resets the network adapter, and re-initializes it with the same set of parameters provided in the previous call to <i>Initialize</i> .
<i>Shutdown</i>	This function resets the network adapter, and leaves it in a state safe for another driver to initialize. The memory buffers assigned in the <i>Initialize</i> call are released. After this call, only the <i>Initialize</i> or <i>Stop</i> calls may be used.
<i>ReceiveFilters</i>	Used to enable and disable the receive filters for the network interface, and if supported, manage the filtered multicast HW MAC (Hardware Media Access Control) address list.
<i>StationAddress</i>	This function can be used to read the current station address, and if supported, allow the station address to be updated.
<i>Statistics</i>	This function can be used to collect statistics from the network interface and allow the statistics to be reset.
<i>MCastIpToMac</i>	This function can be used to map a multicast IP address to a multicast HW MAC address.
<i>NvData</i>	This function can get used to read and write the contents of the NVRAM devices attached to the network interface.
<i>GetStatus</i>	This functions reads the current interrupt status and the list of recycled transmit buffers from the network interface.
<i>Transmit</i>	This function is used to place a packet in the transmit queue.
<i>Receive</i>	This function is used to get a packet from the receive queue along with the status flags that describe the packet type.
<i>WaitForPacket</i>	Event to use with <b>WaitForEvent()</b> to wait for a packet to be received.
<i>Mode</i>	A pointer to the <b>EFI_SIMPLE_NETWORK_MODE</b> data for this device. See “Related Definitions”.



The following data values in **EFI\_SIMPLE\_NETWORK\_MODE** (declared in “Related Definitions”) are read-only and are updated by the code that produces the **EFI\_SIMPLE\_NETWORK** protocol functions. All of these fields must be discovered during driver initialization.

<i>State</i>	Reports the current state of the network interface. When an <b>EFI_SIMPLE_NETWORK</b> driver has initialized a network interface, it is left in the <b>EfiSimpleNetworktopped</b> state.
<i>HwAddressSize</i>	The size, in bytes, of network interfaces's HW address.
<i>MediaHeaderSize</i>	The size, in bytes, of network interfaces's media header.
<i>MacAddressChangeable</i>	<b>TRUE</b> if the HW MAC address can be changed.
<i>MultipleTxSupported</i>	<b>TRUE</b> if the network interface can transmit more than one packet at a time.
<i>CurrentAddress</i>	The current HW MAC address for this network interface.
<i>BroadcastAddress</i>	The current HW MAC address for broadcast packets.
<i>PermanentAddress</i>	The permanent HW MAC address for this network interface.
<i>NvramSize</i>	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 <i>NvramAccessSize</i> .
<i>NvramAccessSize</i>	This is the size that all NVRAM accesses must use. This means the start address for NVRAM reads or writes must be a multiple of this value, and the total length of the read or write operation must also 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 this network interface.
<i>ReceiveFilterMask</i>	Reports the multicast receive filter settings that this network interface supports.
<i>ReceiveFilterSetting</i>	Reports the current multicast receive filter settings.
<i>MCastFilterCnt</i>	Reports the number of multicast address receive filters. If this value is zero, then the multicast address receive filters can not be modified with <b>ReceiveFilters()</b> .
<i>MCastFilter</i>	An array of the current multicast address receive filters addresses.
<i>IfType</i>	The interface type of the network interface. See RFC 1700.

*MediaPresentSupported* **TRUE** if presence of media can be determined. If this field is **FALSE**, then *MediaPresent* can not be used.

*MediaPresent* **TRUE** if media is connected to the network interface. **FALSE** if media is not connected to the network interface.

## 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 produces 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. Please see the PXE Specification for details on these services.

## Related Definitions

```
typedef struct {
    EFI_SIMPLE_NETWORK_STATE      State;
    UINTN                        HwAddressSize;
    UINTN                        MediaHeaderSize;
    BOOLEAN                      MacAddressChangeable;
    BOOLEAN                      MultipleTxSupported;
    EFI_SIMPLE_NETWORK_MAC_ADDR  CurrentAddress;
    EFI_SIMPLE_NETWORK_MAC_ADDR  BroadcastAddress;
    EFI_SIMPLE_NETWORK_MAC_ADDR  PermanentAddress;
    UINTN                        NvramSize;
    UINTN                        NvramAccessSize;
    UINTN                        ReceiveFilterMask;
    UINTN                        ReceiveFilterSetting;
    UINTN                        MCastFilterCount;
    EFI_SIMPLE_NETWORK_MAC_ADDR  *MCastFilter;
    UINT8                        IfType;
    BOOLEAN                      MediaPresentSupported;
    BOOLEAN                      MediaPresent;
} EFI_SIMPLE_NETWORK_MODE;

typedef struct {
    UINT8  Addr[16];
} EFI_SIMPLE_NETWORK_IP_ADDR;

typedef struct {
    UINT8  Addr[16];
} EFI_SIMPLE_NETWORK_MAC_ADDR;
```

```
typedef enum {
    EfiSimpleNetworkStopped,
    EfiSimpleNetworkStarted,
    EfiSimpleNetworkInitialized,
    EfiSimpleNetworkMaxState
} EFI_SIMPLE_NETWORK_STATE;
```

The following is the list of bit mask values for the *ReceiveFilterSetting* field of **EFI\_SIMPLE\_NETWORK\_MODE**. 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
```

The following is the list of interrupt bit mask settings that can be read with the **GetStatus()** function. 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
```



### 15.1.2 EFI\_SIMPLE\_NETWORK.Stop()

#### Summary

Changes the network interface from the started state to the stopped state.

#### 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_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\_NETWORK.Initialize()

#### Summary

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.

#### Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_INITIALIZE) (
    IN EFI_SIMPLE_NETWORK    *This,
    IN UINTN                  ExtraRxBufferSize    OPTIONAL,
    IN UINTN                  ExtraTxBufferSize    OPTIONAL
);
```

#### Parameters

<i>This</i>	A pointer to the <b>EFI_SIMPLE_NETWORK</b> instance.
<i>ExtraRxBufferSize</i>	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.
<i>ExtraTxBufferSize</i>	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_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.







## 15.1.6 EFI\_SIMPLE\_NETWORK.ReceiveFilters()

### Summary

Manages the multicast receive filters of the network interface.

### Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_RECEIVE_FILTERS) (
    IN EFI_SIMPLE_NETWORK      *This,
    IN UINTN                   Enable,
    IN UINTN                   Disable,
    IN BOOLEAN                 ResetMcastFilter,
    IN UINTN                   McastFilterCnt    OPTIONAL,
    IN EFI_SIMPLE_NETWORK_MAC_ADDR *McastFilter    OPTIONAL,
);
```

### Parameters

<i>This</i>	A pointer to the <b>EFI_SIMPLE_NETWORK</b> instance.
<i>Enable</i>	A bit mask of receive filters to enable on the network interface.
<i>Disable</i>	A bit mask of receive filters to disable on the network interface.
<i>ResetMcastFilter</i>	Set to <b>TRUE</b> to reset the contents of the multicast receive filters on the network interface to their default values.
<i>McastFilterCnt</i>	Number of multicast HW MAC addresses in the new <i>McastFilter</i> list. This value must be less than or equal to the <i>McastFilterCnt</i> field of <b>EFI_SIMPLE_NETWORK_MODE</b> . This field is optional if <i>ResetMcastFilter</i> is <b>TRUE</b> .
<i>McastFilter</i>	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 <i>ResetMcastFilter</i> is <b>TRUE</b> .

### Description

This function modifies the current receive filter mask on the network interface. The bits set in *Enable* are enabled 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 this network interface, then **EFI\_INVALID\_PARAMETER** will be returned. The *ReceiveFilterMask* field of **EFI\_SIMPLE\_NETWORK\_MODE** specifies the supported receive filters settings. Please see "Related Definitions" for the list of 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 receive filter mask and multicast receive filter list have been successfully updated on the network interface, then **EFI\_SUCCESS** will be returned.

### Status Codes Returned

EFI_SUCCESS	The multicast receive filter list was updated.
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\_NETWORK.StationAddress()

#### Summary

Allows the station address of the network interface to be modified.

#### Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_STATION_ADDRESS) (
    IN EFI_SIMPLE_NETWORK      *This,
    IN BOOLEAN                  Reset,
    IN EFI_SIMPLE_NETWORK_MAC_ADDR *New    OPTIONAL
);
```

#### Parameters

<i>This</i>	A pointer to the <b>EFI_SIMPLE_NETWORK</b> instance.
<i>Reset</i>	Flag used to reset the station address to the network interface's permanent address.
<i>New</i>	The new station address to be used for this network interface.

#### Description

This function allows the current station address to be modified. If *Reset* is **TRUE**, then the current station address is set to the network interface's permanent address. If *Reset* is **FALSE**, and this network interface allows its station address to be modified, then the current station address is changed to the address specified by *New*. If this 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, then **EFI\_SUCCESS** will be returned.

#### Status Codes Returned

EFI_SUCCESS	The station address was updated on the network interface.
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\_NETWORK.Statistics()

### Summary

Allows the statistics on the network interface to be reset and/or collected.

### Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_STATISTICS) (
    IN EFI_SIMPLE_NETWORK    *This,
    IN BOOLEAN               Reset,
    IN OUT UINTN             *StatisticsSize  OPTIONAL,
    IN OUT EFI_STATISTIC_ENTRY *StatisticsTable  OPTIONAL
);
```

### Parameters

<i>This</i>	A pointer to the <b>EFI_SIMPLE_NETWORK</b> instance.
<i>Reset</i>	Set to <b>TRUE</b> to reset all of the statistics counters on this network interface.
<i>StatisticsSize</i>	On input, the size, in bytes, of the <i>StatisticsTable</i> . On output, the size, in bytes, of the resulting table of statistics. This field is ignored if <i>Reset</i> is <b>TRUE</b> .
<i>StatisticsTable</i>	A pointer to an array of <b>EFI_STATISTIC_ENTRY</b> s that provide all the available statistics counter values for this network interface. This field is ignored if <i>Reset</i> is <b>TRUE</b> . See "Related Definitions" for the declaration of this data structure.

### Description

This function allows statistics on the network interface to be collected. If *Reset* is **TRUE**, then all the statistics counters for the network interface are reset to zero. If *Reset* is **FALSE** and the size of the statistics table specified by *StatisticsSize* is not big enough for all the statistics that are collected by this network interface, then **EFI\_BUFFER\_TOO\_SMALL** is returned, and the *StatisticsSize* value will be set to the size of the buffer needed to collect the table of statistics. If *Reset* is **FALSE**, and the size of the table specified by *StatisticsSize* is big enough for all the statistics collected by this network interface, then the statistics are stored in *StatisticsTable*, the size of *StatisticsTable* is returned in *StatisticsSize*, and **EFI\_SUCCESS** is returned.

### Related Definitions

```
typedef struct {
    EFI_STATISTIC_TYPE    Type;
    UINT32                Counter;
};
```

```
} EFI_STATISTIC_ENTRY;  
  
typedef enum {  
    EfiMaxStatisticType  
} EFI_STATISTIC_TYPE;
```

## Status Codes Returned

EFI_SUCCESS	The statistics were collected from the network interface.
EFI_BUFFER_TOO_SMALL	The <i>StatisticsTable</i> buffer was too small. The current buffer size needed to hold the statistics table is returned in <i>StatisticsSize</i> .
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

Allows a multicast IP address to be mapped 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_SIMPLE_NETWORK_IP_ADDR *IP,
    OUT EFI_SIMPLE_NETWORK_MAC_ADDR *MAC
);
```

#### Parameters

<i>This</i>	A pointer to the <b>EFI_SIMPLE_NETWORK</b> instance.
<i>IPv6</i>	Set to <b>TRUE</b> if the multicast IP address is IPv6 [RFC 2460]. Set to <b>FALSE</b> if the multicast IP address is IPv4 [RFC 791].
<i>IP</i>	The multicast IP address that is to be converted to a multicast HW MAC address.
<i>MAC</i>	The multicast HW MAC address that is to be generated from the multicast IP address <i>IP</i> .

#### Description

This function maps 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_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\_NETWORK.NVData()

#### Summary

Allows read and writes to 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

<i>This</i>	A pointer to the <b>EFI_SIMPLE_NETWORK</b> instance.
<i>ReadWrite</i>	<b>TRUE</b> for read operations, <b>FALSE</b> for write operations.
<i>Offset</i>	Byte offset in the NVRAM device at which to start the read or write operation. This must be a multiple of <i>NvramAccessSize</i> , and less than <i>NvramSize</i> .
<i>BufferSize</i>	The number of bytes to read or write from the NVRAM device. This must also be a multiple of <i>NvramAccessSize</i> .
<i>Buffer</i>	A pointer to the data buffer.

#### Description

This function allows read and write operations to the NVRAM device attached to a network interface. If *ReadWrite* is **TRUE**, then this will be a read operation. If *ReadWrite* is **FALSE**, then this will be a write operation. The byte offset at which to start either operation is specified by *Offset*. *Offset* must be a multiple of the *NvramAccessSize* field in **EFI\_SIMPLE\_NETWORK\_MODE**, and it must have a value between zero and the *NvramSize* field of **EFI\_SIMPLE\_NETWORK\_MODE**. The length of the read or write operation is specified by *BufferSize*. *BufferSize* must also be a multiple of the *NvramAccessSize* field of **EFI\_SIMPLE\_NETWORK\_MODE**, and *Offset* + *BufferSize* must not exceed the *NvramSize* field of **EFI\_SIMPLE\_NETWORK\_MODE**. If any of these conditions is not met, then **EFI\_INVALID\_PARAMETER** will be returned. If this is a read operation, then 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, then 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_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 the network interface.

#### Prototype

```
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_GET_STATUS) (
    IN EFI_SIMPLE_NETWORK      *This,
    OUT UINTN                  *InterruptStatus    OPTIONAL,
    IN OUT UINTN               *TxBufSize          OPTIONAL,
    OUT VOID                   *TxBuf[]           OPTIONAL
);
```

#### Parameters

<i>This</i>	A pointer to the <b>EFI_SIMPLE_NETWORK</b> instance.
<i>InterruptStatus</i>	A pointer to the bit mask of the currently active interrupts. If this is <b>NULL</b> , the interrupt status will not be read from the device. If this is not <b>NULL</b> , 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.
<i>TxBufSize</i>	On entry, the number of entries to be filled in the <i>TxBuf[]</i> array. On exit, the number of entries that were written to <i>TxBuf[]</i> array. If <i>TxBufSize</i> is <b>NULL</b> , or its value is zero, then the recycled transmit buffer status will not be read.
<i>TxBuf</i>	Recycled transmit buffer address array. The network interface will not transmit if its internal recycled transmit buffer array is full. Reading the transmit buffer array does not clear the transmit interrupt. If this is <b>NULL</b> , then the transmit buffer status will not be read.

#### 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*. See "Related Definitions" in Section 15.1 for the bit values for receive interrupts, transmit interrupts, command interrupts, and software interrupts. If *InterruptStatus* is **NULL**, then the interrupt status will not be read. If both *TxBufSize* and *TxBuf* are not **NULL** and the value of *TxBufSize* is not zero, then a list of recycled transmit buffer addresses will be retrieved. On entry, *TxBufSize* specifies the maximum number of entries in *TxBuf[]* that are to be filled. On exit, *TxBufSize* will contain the number of entries of *TxBuf[]* that were actually filled with recycled transmit

buffer addresses. If the status of the network interface is successfully collected, then **EFI\_SUCCESS** will be returned.

### Status Codes Returned

EFI_SUCCESS	The status of the network interface was retrieved.
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\_NETWORK.Transmit()

### Summary

Places a packet in the transmit queue of the 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_SIMPLE_NETWORK_MAC_ADDR *SrcAddr    OPTIONAL,
    IN EFI_SIMPLE_NETWORK_MAC_ADDR *DestAddr   OPTIONAL,
    IN UINT16                   *Protocol      OPTIONAL,
);
```

### Parameters

<i>This</i>	A pointer to the <b>EFI_SIMPLE_NETWORK</b> instance.
<i>HeaderSize</i>	The size, in bytes, of the media header of the packet to transmit through the network interface.
<i>BufferSize</i>	The size, in bytes, of the entire packet including the media header to transmit through the network interface.
<i>Buffer</i>	A pointer to the packet to transmit.
<i>SrcAddr</i>	The source HW MAC address. This parameter should be <b>NULL</b> if the media header has already been initialized.
<i>DestAddr</i>	The destination HW MAC address. This parameter should be <b>NULL</b> if the media header has already been initialized.
<i>Protocol</i>	The type of header to build. This parameter should be <b>NULL</b> if the media header has already been initialized. See RFC 1700 for known types.

### Description

This function places the packet specified by *Buffer* on the transmit queue. *HeaderSize* specifies the size of the media header in *Buffer*, and *BufferSize* specifies the size of *Buffer* in bytes (this is also the size of the entire packet including the media header). If *SrcAddr*, *DestAddr*, and *Protocol* are not **NULL**, then they are used to fill in the media header portion of *Buffer*. If any of these parameters is **NULL**, then it is assumed that the caller has filled in the media header portion of *Buffer*.

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 caller has the option of performing either blocking or non-blocking I/O depending on how the **Transmit()** and **GetStatus()** calls are used.

## Status Codes Returned

EFI_SUCCESS	The packet was placed on the transmit queue.
EFI_NOT_READY	The network interface is too busy to accept this transmit request.
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\_NETWORK.Receive()

#### Summary

Receives a packet from the 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_SIMPLE_NETWORK_MAC_ADDR *SrcAddr    OPTIONAL,
    OUT EFI_SIMPLE_NETWORK_MAC_ADDR *DestAddr   OPTIONAL,
    OUT UINT16                 *Protocol       OPTIONAL
);
```

#### Parameters

<i>This</i>	A pointer to the <b>EFI_SIMPLE_NETWORK</b> instance.
<i>HeaderSize</i>	The size, in bytes, of the media header received on the network interface. If this parameter is <b>NULL</b> , then the media header size will not be returned.
<i>BufferSize</i>	On entry, the size, in bytes, of <i>Buffer</i> . On exit, the size, in bytes, of the packet that was received on the network interface.
<i>Buffer</i>	A pointer to the data buffer to receive both the media header and the data.
<i>SrcAddr</i>	The source HW MAC address. If this parameter is <b>NULL</b> , then the HW MAC source address will not be extracted from the media header.
<i>DestAddr</i>	The destination HW MAC address. If this parameter is <b>NULL</b> , then the HW MAC destination address will not be extracted from the media header.
<i>Protocol</i>	The media header type. If this parameter is <b>NULL</b> , then the protocol will not be extracted from the media header. See RFC 1700 for known types.

## Description

This function retrieves one packet from the receive queue of the 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.

## Status Codes Returned

EFI_SUCCESS	The received data was stored in <i>Buffer</i> , and <i>BufferSize</i> has been updated to the number of bytes received.
EFI_NOT_READY	No packets have been received on the network interface.
EFI_BUFFER_TOO_SMALL	<i>BufferSize</i> is too small for the received packets. <i>BufferSize</i> 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.

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. The system firmware may search the “\EFI” directory of the System Partition to find possible EFI Images that can be loaded. 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 without 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 what size media requires what variant of FAT 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”
- “thisisanexamppldirectory.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**. 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.

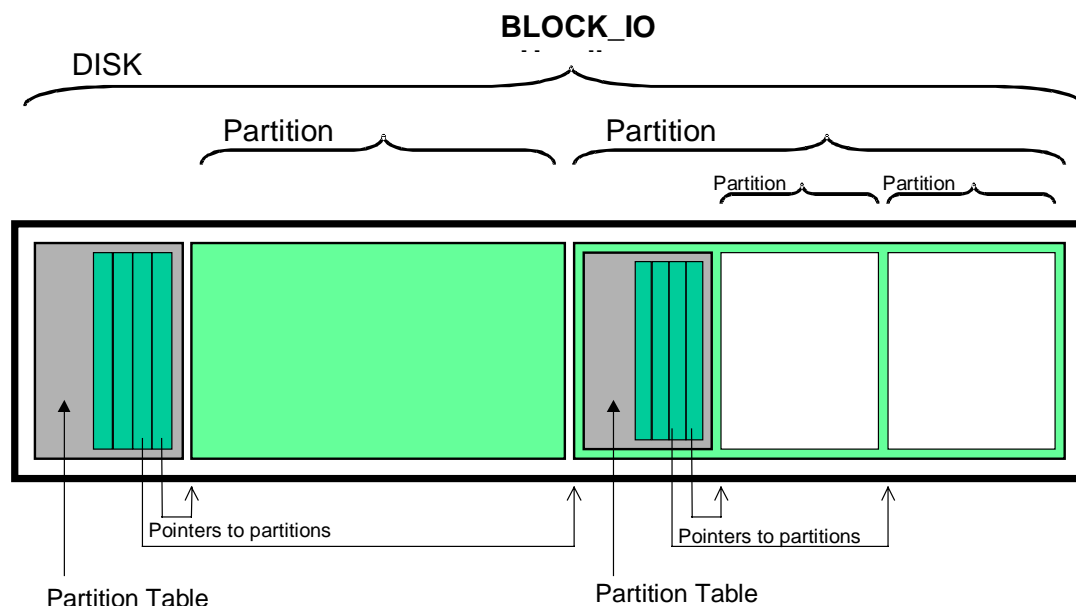
```
\EFI
  \<OS Vendor 1 Directory>
    <OS Loader Image>
  \<OS Vendor 2 Directory>
    <OS Loader Image>
  . . .
  \<OS Vendor N Directory>
    <OS Loader Image>
  \<OEM Directory>
    <OEM Application Image>
  \<BIOS Vendor Directory>
    <BIOS Vendor Application Image>
  \<Third Party Tool Vendor Directory>
    <Third Party Tool Vendor Application Image>
  \BOOT
    <Default Boot Image>
```

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. 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 Image>
```

## 16.2 Partition Discovery

EFI requires the firmware to be able to parse legacy master boot records, the new EFI Partition Table, 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 legacy MBR 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 EFI 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 EFI Partition Table does not allow nesting of EFI Partition Headers. Nesting is not needed since an EFI Partition Header can support an arbitrary number of partitions (the addressability limits of a 64-bit LBA is the limiting factor).

### 16.2.1 Extensible Firmware Interface Partition Header

EFI defines a new partitioning scheme that must be supported by EFI firmware. The following list outlines the advantages of using the EFI Partition Table over the legacy MBR partition table:

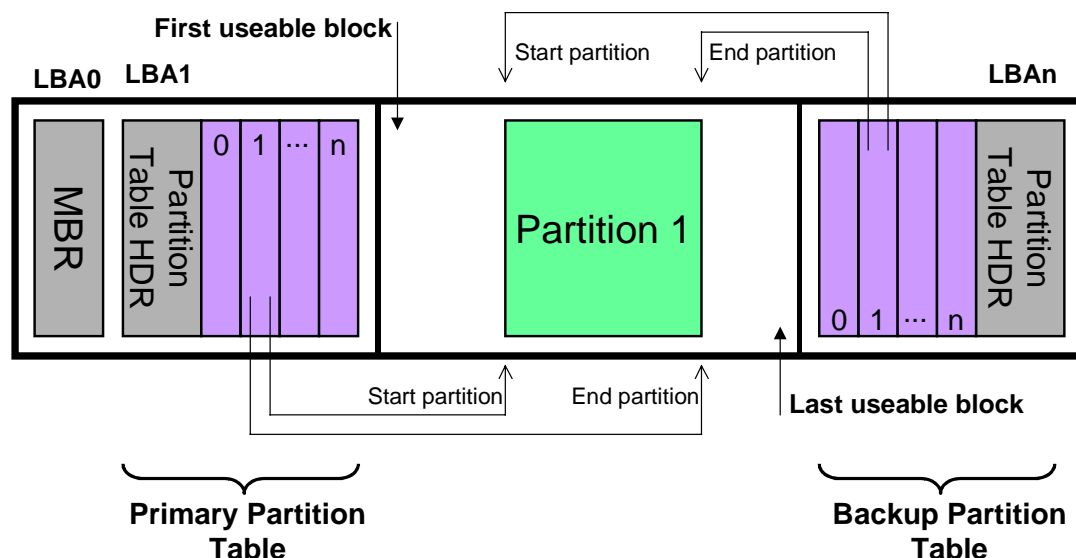
- Logical Block Addressing is 64-bits.
- Supports many partitions.
- Uses a primary and backup table for redundancy.
- Uses version number and size fields for future expansion.
- Uses CRC32 fields for improved data integrity.
- Defines a GUID for uniquely identifying each partition.
- Uses a GUID and 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 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 Partition Table Header contains a header size field that is used in calculating the CRC32 that confirms the integrity of the Partition Table Header. While the Partition Table Header's size may increase in the future it can not span more than one block on the device.

Two Partition Table Header structures are stored on the device: the primary and the backup. The primary Partition Table Header must be located in block 1 of the logical device, and the backup Partition Table Header must be located in the last block of the logical device. Within the Partition Table Header there are the MyLBA and AlternateLBA fields. The MyLBA field contains the logical block address of the Partition Table Header itself, and the AlternateLBA field contains the logical block address of the other Partition Table Header. For example, the primary 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 Partition Table Header's fields would be reversed.

The Partition Table Header defines the range of logical block addresses that are usable by Partition Entries. This range is defined to be inclusive of FirstUsableLBA through LastUsableLBA on the logical device. All data stored on the volume must be stored between the FirstUsableLBA through LastUsableLBA, and only the data structures defined by EFI to manage partitions may reside outside of the usable space. The value of DiskGUID is a GUID that uniquely identifies the entire Partition Table Header and all its associated storage. This value can be used to uniquely identify the disk. The start of the Partition Entry array is located at the logical block address PartitionEntryLBA. The size of a Partition Entry element is defined in the Partition Table Header. There is a 32-bit CRC of the Partition Entry array that is stored in the Partition Table Header in PartitionEntryArrayCRC. The size of the Partition Entry array is the PartitionEntrySize multiplied by NumberOfPartitionEntries. When a Partition Entry is updated the PartitionEntryArrayCRC

must be updated. When the PartitionEntryArrayCRC is updated the Partition Table Header CRC must also be updated, since the PartitionEntryArrayCRC is stored in the Partition Table Header.



**Figure 16-2. EFI Partitioning Scheme**

The primary Partition Entry array must be located after the primary Partition Table Header and end before the FirstUsableLBA. The backup Partition Entry array must be located after the LastUsableLBA and end before the backup Partition Table Header. Therefore the primary and backup Partition Entry arrays are stored in separate locations on the disk. Partition Entries define a partition that is contained in a range that is contained within the usable space declared by the Partition Table Header. Zero or more Partition Entries may be in use in the Partition Entry array. Each defined partition must not overlap with any other defined partition. If all the fields of a Partition Entry are zero the entry is not in use. A minimum of 16,384 bytes of space must be reserved for the Partition Entry array. Thus the First useable block must start at an LBA greater than or equal to 34.

**Table 16-1. EFI 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 EFI Partition Table Header.
HeaderCRC32	16	4	CRC32 checksum for the EFI Partition Table Header structure. The range 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 Partition Table Header.
FirstUsableLBA	40	8	The first usable logical block that may be contained in a Partition Entry.
LastUsableLBA	48	8	The last usable logical block that may be contained in a Partition Entry.
DiskGUID	56	16	GUID that can be used to uniquely identify the disk.
PartitionEntryLBA	72	8	The starting LBA of the Partition Entry array.
NumberOfPartitionEntries	80	4	The number of Partition Entries in the Partition Entry array.
SizeOfPartitionEntry	84	4	The size, in bytes, of each the Partition Entry structure in the Partition Entry array. Must be a multiple of 8.
PartitionEntryArrayCRC32	88	4	The CRC32 of the Partition Entry array. Starts at Partition Entry LBA and is <i>NumberOfPartitionEntries * SizeOfPartitionEntry</i> 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 Partition Table is valid:

- Check the Partition Table Signature
- Check the Partition Table CRC
- Check that the MyLBA entry points to the LBA that contains the Partition Table
- Check the CRC of the Partition Entry Array

If the EFI Partition Table is the primary table, stored at LBA 1:

- Check the AlternateLBA to see if it is a valid EFI Partition Table

If the primary Partition Table is corrupt:

- Check the last LBA of the device to see if it has a valid Partition Table
- If valid backup Partition Table found, restore primary Partition Table.

Any software that updates the primary Partition Table Header must also update the backup Partition Table Header. The order of the update of the Partition Table Header and its associated Partition Entry array is not important, since all the CRCs are stored in the Partition Table Header. However, the primary Partition Table Header and Partition Entry array must always be updated before the backup.

If the primary Partition Table is invalid the backup Partition Table is located on the last logical block on the disk. If the backup Partition Table is valid it must be used to restore the primary Partition Table. If the primary Partition Table is valid and the backup Partition Table is invalid software must restore the backup Partition Table. If both the primary and backup Partition Table is corrupted this block device is defined as not having a valid EFI Partition Header.

The primary and backup Partition Tables must be valid before an attempt is made to grow the size of a physical volume. This is due to the Partition Table recovery scheme depending on locating the backup 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 Partition Table must be moved to the end of the volume and the primary and backup Partition Table Headers must be updated to reflect the new volume size.

**Table 16-2. EFI 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 Partition Entry is created. The partition Entry is created when ever the <i>NumberOfPartitionEntries</i> in the EFI 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 Partition Table Header defines the size of a Partition Entry. The Partition Entry starts in the first byte of the 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 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.

**Table 16-3. Defined EFI Partition Entry Type GUID**

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 EFI 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.
Bits 1-63	Undefined and must be zero. Reserved for expansion by future versions of the EFI specification.

## 16.2.2 ISO-9660 and El Torito

ISO-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.

**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.
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 EFI Partitions

The EFI partition structure does not support nesting of partitions. However it is legal to have a legacy Master Boot Record nested inside an EFI Partition.

An EFI Partition header is preceded by a legacy MBR in the first LBA of the disk to maintain compatibility with existing tools that do not understand EFI Partition structures. The MBR that precedes an EFI Partition Header is shown in Table 16-7, and this MBR represents the entire space used on the disk by the EFI Partitions including all headers. If the EFI partition is larger than a partition that can be represented by the 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. Required Legacy Master Boot Record Entry to Precede an EFI Partition 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 0xFFFFFFFF 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 0xFFFFFFFF 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 of the EFI Specification describes how booting from different types of removable media is handled. In general the rules are consistent regardless of the media being removable or not and the media's physical type.

### 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.

Bootting 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.

Once the “\EFI” directory is found, the boot manager then checks each Vendor GUID directory for any EFI applications. The first application that is found to be compatible with the platform is then implicitly loaded. To build removable media that load deterministically, the vendor should supply media that contain exactly one Vendor GUID directory with exactly one image of each machine type that the vendor supports.

### 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. To support EFI booting, the diskette would contain an EFI directory that contains one Vendor GUID directory. The vendor GUID directory would contain one EFI application image of each machine type that the vendor supports. Then when the EFI Boot Manager is configured to boot from the floppy, it would load the first EFI image that the platform supports.

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<sup>†</sup> Zip<sup>†</sup>, Fujitsu MO, or MKE LS-120/SuperDisk<sup>†</sup>.

### 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.



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 to the end of 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 this would be not loading an EFI driver if booting failed the first time the driver was loaded. An other 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 what file on that hardware device contains the EFI image that is to be loaded. The NVRAM also contains 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.

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.

The boot manager must call *LoadImage()* which supports at least *SIMPLE\_FILE\_PROTOCOL* and *LOAD\_FILE\_PROTOCOL* for resolving load options. The complete list of protocols that the *LoadImage()* function can support loading images from is available via the *LocateProtocol()* function. The device referenced by a load option must support one of the boot manager's boot device protocols in order for the load option to function.

If the boot image is not loaded via *LoadImage()* the boot manager is required to check for a default media boot. This occurs when the device path of the boot image points directly to a *BLOCK\_IO* device and does not specify the exact file to load. The boot manager must locate the handle(s) for the *BLOCK\_IO* device that support the *SIMPLE\_FILE\_SYSTEM* protocol that are present and check for the EFI directory. If found, the boot manager then checks each Vendor GUID directory for any EFI applications. The first application found to be compatible with the platform is then implicitly loaded. To build removable media that load deterministically the vendor should supply a media that contains exactly 1 Vendor GUID directory with exactly 1 image of each machine type that the vendor supports. The default media boot case of a protocol other than *BLOCK\_IO* is handled by the *LOAD\_FILE\_PROTOCOL* for the target device path and does not need to be handled by the boot manager.

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;
    CHAR16                Description[];
    EFI_DEVICE_PATH        FilePath[];
    UINT8                 OptionalData[];
} EFI_LOAD_OPTION;
```

## Parameters

<i>Attributes</i>	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”.
<i>Description</i>	The user readable description for the load option. This field ends with a Null Unicode character.
<i>FilePath</i>	An EFI device path that describes the device and location of the Image for this load option. The <i>FilePath</i> is specific to the device type. This field 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.
<i>OptionalData</i>	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. 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	The device path of the default input console.
ConOut	NV, BS	The device path of the default output console.
ErrOut	NV, BS	The device path of the default error output device.
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, RT	The boot option for the next boot only.
Driver####	NV, BS	A driver load option. #### is a printed hex value.
DriverOrder	NV, BS	The ordered driver load option list.
SerialNumber	NV, BS	The system's serial number.
SystemGUID	NV, BS	The system's Guaranteed Unique Identifier.



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.

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.

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.

The *SerialNumber* variable contains a Null-terminated Unicode string that represents the system's serial number.

The *SystemGUID* variable is a 128 bit value that contains the system's Guaranteed Unique Identifier. Please see the *Wired for Management Specification* for details.

## 17.3 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.

### 17.3.1 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.

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.

### 17.3.2 Boot via LOAD\_FILE Protocol

When booting via the **LOAD\_FILE** protocol, the *FilePath* will point to 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.

#### 17.3.2.1 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 (see Chapter 14).

#### 17.3.2.2 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 media as long as it is possible to find a partition on the boot media. 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.

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 media.

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 Version 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; IA-64; and intermediate byte stream. The IA-32 and IA-64 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. This is done by verifying that no handle in the system supports the exact device path of the device to be controlled. Therefore when the driver installs, it must add the **DEVICE\_PATH** protocol to the handle that is controlling the device. This would either be overloaded onto the image handle, or a new handle that the driver has allocated for this task.

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.

## GUID and Time Formats

All EFI GUIDs (Globally Unique Identifiers) have the format described in Appendix J of the *Wired for Management Baseline* specification. This document references the format of the GUID, but implementers must reference the Wired for Management specifications for algorithms to generate GUIDs. The following table defines the format of an EFI GUID (128 bits).

**Table A-1. EFI GUID Format**

Mnemonic	Byte Offset	Byte Length	Description
TimeLow	0	4	The low field of the timestamp.
TimeMid	4	2	The middle field of the timestamp.
TimeHighAndVersion	6	2	The high field of the timestamp multiplexed with the version number.
ClockSeqHighAndReserved	8	1	The high field of the clock sequence multiplexed with the variant
ClockSeqLow	9	1	The low field of the clock sequence.
Node	10	6	The spatially unique node identifier. This can be based on any IEEE 802 address obtained from a network card. If no network card exists in the system, a cryptographic-quality random number can be used.

All EFI time is stored in the format described by Appendix J of the *Wired for Management Baseline Specification*. 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.





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
0x15	Function 11	CSI O A	ESC [ O A	0x85
0x16	Function 12	CSI O B	ESC [ O B	0x86
0x17	Escape	CSI	ESC	0x01

## B.2 SIMPLE\_TEXT\_OUTPUT

Table B-2 defines how the programmatic methods of the `SIMPLE_TEXT_OUTPUT` 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)

<b>PCANSI Codes</b>	<b>ANSI X3.64 Codes</b>	<b>Description</b>
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 [ <i>row</i> ; <i>col</i> H	CSI <i>row</i> ; <i>col</i> H	Set cursor position to <i>row</i> ; <i>col</i> . <i>Row</i> and <i>col</i> 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.

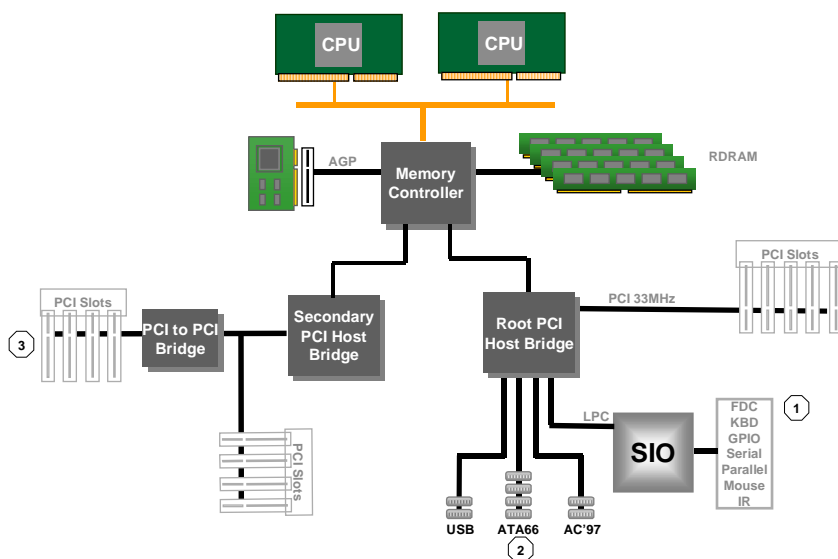
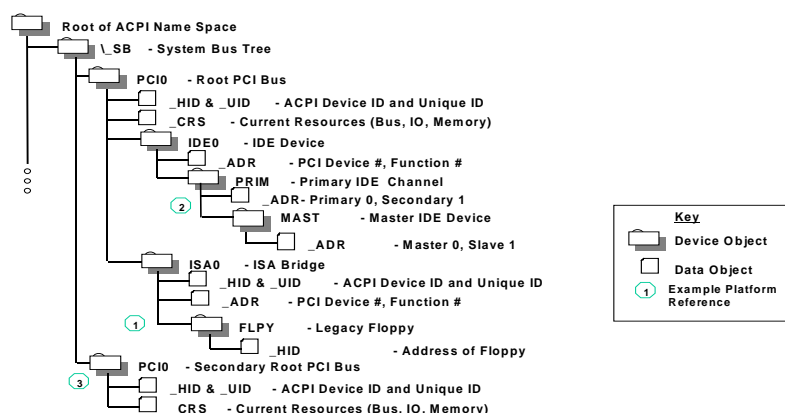


Figure C-1. Example Computer System

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*.



**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	<b>Generic Device Path Header</b> – 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	<b>Generic Device Path Header</b> – Type Hardware Device Path
D	1	0x01	Sub type PCI Device Path
E	2	0x08	Length
10	1	0x00	PCI Function
11	1	0x10	PCI Device
12	1	0x02	<b>Generic Device Path Header</b> – 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	<b>Generic Device Path Header</b> – 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	<b>Generic Device Path Header</b> – 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	<b>Generic Device Path Header</b> – Type Hardware Device Path
D	1	0x01	Sub type PCI Device Path
E	2	0x08	Length
10	1	0x01	PCI Function
11	1	0x10	PCI Device
12	1	0x03	<b>Generic Device Path Header</b> – Messaging Device Path
13	1	0x01	Sub type – ATAPI Device Path
14	2	0x08	Length
16	1	0x00	Primary =0, Secondary = 1
17	1	0x00	Master = 0, Slave = 1
18	2	0x0000	LUN
1A	1	0xFF	<b>Generic Device Path Header</b> – 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	<b>Generic Device Path Header</b> – Type ACPI Device Path
1	1	0x01	Sub type – ACPI Device Path
2	2	0x0C	Length
4	4	0x41D0, 0x0A03	<code>_HID PNP0A03</code> – 0x41D0 represents a compressed string 'PNP' and is in the low order bytes
8	4	0x0001	<code>_UID</code>
C	1	0x01	<b>Generic Device Path Header</b> – Type Hardware Device Path
D	1	0x01	Sub type PCI Device Path
E	2	0x08	Length
10	1	0x00	PCI Function for PCI to PCI bridge
11	1	0x0c	PCI Device for PCI to PCI bridge
12	1	0x03	<b>Generic Device Path Header</b> – 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	<b>Generic Device Path Header</b> – 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.

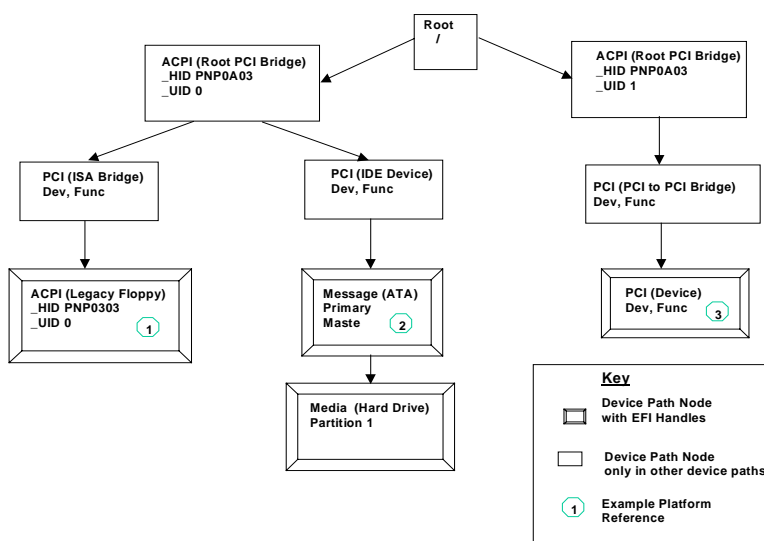


Figure C-3. EFI Device Path Displayed As a Name Space



# D

## Status Codes

EFI interfaces return an **EFI\_STATUS** code. Tables D-1, D-2, and D-3 list these codes for success, errors, and warnings, respectively. Error codes also have their highest bit set, so all error codes have negative values.

**Table D-1. EFI\_STATUS Success Codes**

Mnemonic	Value	Description
EFI_SUCCESS	0	The operation completed successfully.

**Table D-2. EFI\_STATUS Error Codes**

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.
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 inconsistency 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 media to perform the operation.
EFI_MEDIA_CHANGED	13	The media 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.

continued

**Table D-2. EFI\_STATUS Error Codes** (continued)

Mnemonic	Value	Description
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.

**Table D-3. EFI\_STATUS Warning Codes**

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 List

The following table lists all EFI functions alphabetically. 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-1. Alphabetic Function List**

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.
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	Used by EFI components to convert 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.
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.

continued

**Table E-1. Alphabetic Function List** (continued)

Function Name	Service or Protocol	Sub-Service	Function Description
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.
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.
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.

continued



**Table E-1. Alphabetic Function List** (continued)

Function Name	Service or Protocol	Sub-Service	Function Description
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
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.
MetaMatch	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.

continued

**Table E-1. Alphabetic Function List** (continued)

Function Name	Service or Protocol	Sub-Service	Function Description
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.
ReceiveFilters	Simple Input Protocol		Manages the multicast receive filters of the network interface.
RegisterProtocolNotify	Boot Services	Protocol Handler Services	Registers for protocol interface installation notifications
ReinstallProtocolInterface	Boot Services	Protocol Handler Services	Replaces a protocol interface.
Reset	Block I/O Protocol		Resets the block device hardware.
Reset	Serial I/O Protocol		Resets the hardware device.

continued

**Table E-1. Alphabetic Function List** (continued)

Function Name	Service or Protocol	Sub-Service	Function Description
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.
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.

continued

**Table E-1. Alphabetic Function List** (continued)

Function Name	Service or Protocol	Sub-Service	Function Description
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.
StrLwr	Unicode Collation Protocol		Converts all the Unicode characters in a Null-terminated Unicode string to lower case Unicode characters.

continued

**Table E-1. Alphabetic Function List** (continued)

Function Name	Service or Protocol	Sub-Service	Function Description
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.



**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.

**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.

**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 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.

## CIM (Common Information Model)

A schema into which platform configuration data is mapped by higher-level software.

## 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 user.

## 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.

## 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.

## Device Path

An identifier used to locate and differentiate devices on the platform in the boot services environment.

## Device Path Protocol

A protocol that is used during boot services to provide the information needed to construct and manage device paths.

## 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.



**DMI**

Desktop Management Interface.

**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 Image**

An executable file stored in a file system that complies with this specification. Images may be drivers, applications or OS loaders.

**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 IA-64 architecture.

**Event Services**

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

**File System Protocol**

A protocol that is used during boot services to obtain file-based access to a device.

**Firmware**

Any software that is included in read-only memory (ROM).

**GUID** (Guaranteed 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**.

**Handle**

See **Device Handle**.

**IA-32**

The 32-bit architecture implemented in the Pentium and Pentium Pro processors.

**IA-64**

Includes the IA-32 architecture and the high performance 64-bit EM architecture extension.

**Image Services**

The set of functions used to manage EFI images. Includes LoadImage(), StartImage(), UnloadImage(), Exit(), ExitBootServices(), and EFI\_IMAGE\_ENTRY\_POINT.

**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.

**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.

**MCA** (Machine Check Abort)

The system management and error correction facilities built into the IA-64 processor family.

**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.

## Miscellaneous Services

Various functions that are needed to support the EFI environment. Includes **ResetSystem()**, **Stall()**, **SetWatchdogTimer()**, **GetNextMonotonicCount()**, and **GetNextHighMonotonicCount()**.

## Page Memory

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

## Pool Memory

A set of contiguous bytes. A pool begins on, but need not end on, a page 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()**.

## Partition

A section of a block device treated as a logical whole. For fixed disks, the master boot record defines the number and size of partitions present on a given disk.

## PC-AT

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

## Preboot Execution Environment

See **PXE**.

## Protocol

The set of 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()`, `LocateDevicePath()`, and `LocateProtocol()`.

## 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

Refers to the “Preboot Execution Environment”. For more information, see the *Preboot Execution Environment (PXE) Specification*.

## Runtime Services

Interfaces that provide access to underlying platform specific hardware that may be useful during OS runtime, such as timers. These services are available during the boot process but also persist after the OS loader terminates boot services.

## SAL (System Abstraction Layer)

Firmware that abstracts platform implementation differences, and provides the basic platform software interface to all higher level software.

## Serial I/O Protocol

A protocol that is used during boot services to abstract byte stream devices.

## SMBIOS (System Management BIOS)

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.

## 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.

**System Abstraction Layer**

See SAL.

**System Management BIOS**

See SMBIOS.

**System Partition**

An optional partition that may appear on a mass storage device (or any other media supporting the BLOCK\_IO protocol). If such a partition exists separate from native OS format partitions, the structure of the file system on that partition must comply with the structures defined in this specification.

**String**

All strings in this specification are implemented in Unicode.

**Task Priority Level**

See TPL.

**Task Priority Services**

The set of functions used to manipulate task priority levels. Includes RaiseTPL() and RestoreTPL().

**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**

Task Priority Level. The boot services environment exposes three task priority levels: “normal”, “callback”, and “notify”.

**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 lexical comparison functions on Unicode strings for given languages.

**Universal Serial Bus**

See USB.

**USB (Universal Serial Bus)**

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.

**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 (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.

**Wired for Management**

See **WfM**.

## A

ACPI, 7  
 Advanced Configuration and Power Interface specification, 7. *See also* related information  
 AllocatePages() Boot Services Function, 41  
 AllocatePool() Boot Services Function, 49  
 ANSI 3.64 terminals, and  
     SIMPLE\_TEXT\_OUTPUT, 306  
 Application, EFI, 104  
 attributes, of specification, 3

## B

bibliography, 5  
 BIOS Boot Specification Device Path, other rules, 130  
 BIOS, Boot Specification Device, 126  
 Block Size, definition of, 327  
 BLOCK\_IO Protocol, 165, 175  
 BLOCK\_IO.FlushBlocks Function, 173  
 BLOCK\_IO.ReadBlocks Function, 169, 177  
 BLOCK\_IO.Reset Function, 168  
 BLOCK\_IO.WriteBlocks Function, 171, 178  
 Boot Device, definition of, 327  
 boot manager, 293  
 Boot Manager, definition of, 327  
 boot manager, firmware, 293  
 boot process, illustration of, 13  
 boot process, overview, 13  
 boot sequence, 293  
 Boot Service Function  
     LoadImage(), 65  
     LocateHandle(), 58, 61  
     RegisterProtocolNotify(), 57  
 Boot Services Function  
     AllocatePages(), 41  
     AllocatePool(), 49  
     Exit(), 70  
     ExitBootServices(), 72  
     FreePages(), 44

FreePool(), 50  
 GetMemoryMap(), 45  
 SetWatchdogTimer(), 94  
 StartImage(), 67

### Boot Services Functions

CloseEvent(), 30  
 CreateEvent(), 26  
 RaiseTPL(), 35  
 RestoreTPL(), 37  
 SetTimer(), 33  
 SignalEvent(), 31

Boot Services, definition of, 328

boot services, general execution  
     list, 91

Boot Specification Device, BIOS, 126

booting sequence, illustration of, 13

booting, network, 298

booting, overview of, 13

## C

calling conventions, 15  
 calling conventions, IA-32, 17  
 calling conventions, IA-64, 18  
 calling conventions, IA-64 procedures, 18  
 CloseEvent() Boot Services Function, 30  
 Console (), 305, 319  
 Console, overview, 145  
 ConsoleIn, definition, 146, 328  
 ConsoleOut, 151  
 contents, document, 2  
 conventions, document, 11  
 CreateEvent() Boot Services Function, 26

## D

data types, EFI, 16  
 design overview, 8  
 Device Handle to Protocol Handler Mapping,  
     figure, 52  
 Device Handle, definition of, 328

- Device I/O Protocol, 131
- Device I/O, overview, 131
- Device Path, hardware, 115
- Device Path protocol, 111
- Device Path structures, generic, 113
- Device Path, definition of, 328
- Device Path, examples, 309
- Device Path, hardware
  - ACPI, 117
  - memory-mapped, 116
  - PCCARD, 116
  - PCI, 115
  - vendor, 116
- Device Path, media, 123
  - CD-ROM, 125
  - file path, 126
  - hard drive, 123
  - Vendor-Defined, 125
- Device Path, messaging, 117
  - 1394, 119
  - ATAPI, 118
  - FibreChannel, 118
  - I2O, 119
  - NGIO, 121, 122
  - SCSI, 118
  - TCP/IP, 120, 121
  - USB, 119
  - Vendor-Defined, 123
- Device Path, other rules, 130
- Device Path, overview, 111
- Device Path, structures, 113
  - End, 115
- DEVICE\_IO Mem(), Io(), Pci() functions, 135
- DEVICE\_IO protocol, 132
- DEVICE\_IO.AllocateBuffer() function, 141
- DEVICE\_IO.Flush() function, 143
- DEVICE\_IO.Map() function, 138
- DEVICE\_IO.PciDevicePath() function, 137
- DEVICE\_IO.Unmap() XE
  - "function:DEVICE\_IO.Unmap()"
  - function, 102, 140
- DEVICE\_IO\_PROTOCOL, 131
- DEVICE\_PATH protocol, 112
- Driver, EFI, 104

## E

- EFI Application, definition of, 329
- EFI Console, definition, 145
- EFI Driver, 104
- EFI Driver, definition of, 329
- EFI File, definition of, 329
- EFI Image, 99
- EFI Image Handoff State, 105
- EFI Image Header, 103
- EFI Image, definition of, 329
- EFI Images, 103
- EFI Images, introduction, 103
- EFI Images, loading, 13
- EFI interfaces, purpose, 14
- EFI OS Loader, 104
- EFI OS loader, definition of, 329
- EFI runtime service functions, 15
- EFI services, 14
- EFI System Table, definition, 105
- EFI\_FILE\_SYSTEM\_INFO, 198
- EFI\_GENERIC\_FILE\_INFO, 196
- EFI\_IMAGE\_ENTRY\_POINT, 69
- EFI\_LOAD\_OPTION descriptor, definition of, 295
- EFI\_STATUS, return codes, 317
- EFIApplication, 104
- error codes, EFI\_STATUS, 317, 318
- Event Services, 24
- examples, Device Path, 309
- Exit() Boot Services Function, 70
- ExitBootServices() Boot Services Function, 72

## F

- File System Protocol, 179
- File system. format, 279
- FILE\_IO.Close, 187
- FILE\_IO.Delete, 188
- FILE\_IO.GetInfo, 193
- FILE\_IO.GetPosition, 192
- FILE\_IO.Open, 181, 184
- FILE\_IO.Read, 189
- FILE\_IO.SetInfo, 194, 195
- FILE\_IO.SetPosition, 191
- FILE\_IO.Write, 190



firmware boot manager, 293  
 firmware menu, 13  
 formats, of different media, 290  
 FreePages() Boot Services Function, 44  
 FreePool() Boot Services Function, 50  
 function  
   —  
     SIMPLE\_TEXT\_OUTPUT\_INTERFACE.TestString, 157  
     DEVICE\_IO.Flush(), 143  
     DEVICE\_IO.Map(), 138  
     DEVICE\_IO.PciDevicePath(), 137  
     DEVICE\_IO.Unmap(), 102, 140  
     PXE\_IO.Startup(), 232, 234, 235, 237, 240, 243, 245, 248, 249, 250, 251, 252, 253  
     SIMPLE\_INPUT\_INTERFACE.ReadKeyStroke, 150, 207, 208, 210, 211, 212, 213, 217, 218, 220, 221, 222, 223, 324  
     SIMPLE\_INPUT\_INTERFACE.Reset, 149  
     SIMPLE\_TEXT\_OUTPUT\_INTERFACE.OutputString, 155  
     SIMPLE\_TEXT\_OUTPUT\_INTERFACE.ClearScreen, 162  
     SIMPLE\_TEXT\_OUTPUT\_INTERFACE.EnableCursor, 164  
     SIMPLE\_TEXT\_OUTPUT\_INTERFACE.QueryMode, 158  
     SIMPLE\_TEXT\_OUTPUT\_INTERFACE.Reset, 154  
     SIMPLE\_TEXT\_OUTPUT\_INTERFACE.SetAttribute, 160  
     SIMPLE\_TEXT\_OUTPUT\_INTERFACE.SetCursorPosition, 163  
 functions, DEVICE\_IO Mem(), Io(), Pci(), 135

## G

GetMemoryMap() Boot Services Function, 45  
 GetNextVariableName() Runtime Function, 76  
 GetTime() Runtime Function, 81  
 GetVariable() Runtime Function, 74  
 globally unique identifier, definition of, 330  
 Globally Unique Identifiers, algorithms, 203  
 Globally Unique Identifiers, format, 203

globally-defined variables, 296  
 glossary, 327  
 goals of specification, 3  
 GUID, algorithms, 203  
 GUID, definition of, 330  
 GUID, format, 203

## H - I

Handle, definition of, 330  
 HandleProtocol() Function, 60  
 handoff state, IA-32, 109  
 handoff state, IA-64, 110  
 IA-32, handoff state, 109  
 IA-64  
   firmware specifications, 8. *See also* related information  
   platforms, 8  
   requirements, related to this specification, 8  
 IA-64, handoff state, 110  
 Image Boot Service Interfaces, list, 64, 73  
 Image Handoff State, 105  
 Image Header, 103  
 Image Services, 63  
 Image, EFI, contents, 279  
 information, resources, 5  
 InstallProtocolHandler() Function, 53  
 Intel Architecture Platform Architecture, definition of, 330  
 interfaces, EFI  
   general categories of, 15  
 introduction, 1

## L - M

legacy systems, support of, 10  
 little endian, 11  
 LOAD\_FILE Protocol, 199  
 LOAD\_FILE.LoadFile() Function, 200  
 LOADED\_IMAGE Protocol, 99  
 Loader, EFI, 104  
 LoadImage() Boot Service Function, 65  
 LocateHandle() Boot Service Function, 58, 61  
 Master Boot Record, 123  
 MBR. *See* Master Boot Record  
 media formats, 290

- media type, finding partition
  - CD-ROM and DVD-ROM, 287, 291
  - diskette, 291
  - hard drive, 291
- media types, finding partition, future types, 298
- Memory Allocation, Overview, 38
- memory map, 38
- Memory Map, definition of, 331
- memory type, usage after
  - ExitBootServices(), 38
- migration requirements, 10
- migration, from legacy systems, 10
- Miscellaneous services, 91
- N - O**
- network booting, 298
- operating system loader, definition of, 329
- organization, document, 2
- OS loader, definition of, 329
- overview of design, 8
- overview, of boot process, 13
- P**
- Partition, definition of, 331
- PCANSI terminals, and
  - SIMPLE\_TEXT\_OUTPUT, 306
- PCI Address, structure, 136
- PE32+ image format, 103
- Plug and Play Option ROMs
  - and boot services, 14
- prerequisite specifications, 7
- protocol
  - Device Path, 111
  - LOADED\_IMAGE, 99
  - PXE\_IO, 225
  - SIMPLE\_INPUT\_INTERFACE, 146, 215
  - SIMPLE\_TEXT\_OUTPUT\_INTERFACE, 151
- Protocol Handler Services, 51
- Protocol Handler, definition of, 332
- Protocol Interface, definition of, 332
- protocol, DEVICE\_IO, 132
- protocols, 18

- Protocols
  - Block\_IO, 165, 175
  - LOAD\_FILE, 199
  - SIMPLE\_FILE\_SYSTEM, 179
- protocols, code illustrating, 19
- protocols, EFI, list of, 20
- purpose of specification, 1
- PXE\_IO protocol, 225
- PXE\_IO protocol, definition, 225
- PXE\_IO.Startup() function, 232, 234, 235, 237, 240, 243, 245, 248, 249, 250, 251, 252, 253
- R**
- RaiseTPL() Boot Services Function, 35
- references, 5
- RegisterProtocolNotify() Boot Service Function, 57
- related information, 5
- ResetSystem() Runtime Function, 92
- RestoreTPL() Boot Services Function, 37
- rules, for Device Path generation, 127
  - \_UID, 128
  - ACPI\_ADR, 128
  - ACPI\_HID, 128
  - hardware vs. messaging, 129
- rules, for Device Path, media, 129
- Runtime Function
  - GetNextVariableName(), 76
  - GetTime(), 81
  - GetVariable(), 74
  - ResetSystem(), 92
  - SetVariable(), 78
  - SetWakeupTime(), 86
- runtime service functions, 15
- runtime services, 15
- Runtime Services, definition of, 332
- S**
- Scan Codes, for
  - SIMPLE\_INPUT\_INTERFACE, 146
- scope of specification, 1
- services
  - miscellaneous, 91
  - variable, 73

- services, EFI, 14
- Services, introduction, 23
- SetTimer() Boot Services Function, 33
- SetVariable() Runtime Function, 78
- SetWakeupTime() Runtime Function, 86
- SetWatchdogTimer() Boot Services Function, 94
- SignalEvent() Boot Services Function, 31
- SIMPLE\_FILE\_SYSTEM Protocol, 179
- SIMPLE\_INPUT\_INTERFACE protocol, 146, 215
- SIMPLE\_INPUT\_INTERFACE.ReadKeyStroke function, 150, 207, 208, 210, 211, 212, 213, 217, 218, 220, 221, 222, 223, 324
- SIMPLE\_INPUT\_INTERFACE.Reset function, 149
- SIMPLE\_TEXT\_OUTPUT protocol, implementation, 305, 319
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE protocol, 151
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE.ClearScreen function, 162
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE.EnableCursor function, 164
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE.OutputString function, 155
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE.QueryMode function, 158
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE.Reset function, 154
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE.SetAttribute function, 160
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE.SetCursorPosition function, 163
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE.SetMode function, 159
- SIMPLE\_TEXT\_OUTPUT\_INTERFACE.TestString function, 157
- specifications, other, 7
- specifications, prerequisite, 7
- Stall() Boot Services Function, 96
- StandardError, 151
- StandardError, definition of, 332
- StartImage() Boot Services Function, 67
- status codes, EFI, 317

- storage device types, allowed by specification, 9
- String, definition of, 333
- structures, Device Path, 113
- success codes, EFI status, 317
- System Partition, definition, 279, 280
- System Partition, definition of, 333
- System Partition, EFI, 279

## T - U

- target audience, 5
- Task Priority Levels, 24
- terminology, definitions, 327
- Time services, 80
- time, format, 203
- TPL, 24
- typographic conventions, 11
- Unicode control characters, supported, 146
- Unicode, definition of, 333
- UninstallProtocolHandler() Function, 55
- URLs, of related information, 5

## V - W

- variable
  - Boot####, 297
  - BootNext, 297
  - BootOrder, 297
  - ConIn, 297
  - ConOut, 297
  - Driver####, 297
  - DriverOrder, 297
  - ErrOut, 297
  - Lang, 297
  - LangCodes, 297
  - Timeout, 297
- Variable services, 73
- variables, globally-defined, 296
- variables, with architecturally-defined meanings, 296
- warning codes, EFI\_STATUS, 318
- Watchdog timer, definition of, 334
- web sites, 5
- WfM. *See* Wired for Management specification
- Wired for Management specification, 8. *See also* related information

