Driver Writer's Guide For UEFI 2.0

June 13, 2008

Revision 0.96

Acknowledgements

Revision History

Revision	Revision History	Date
0.31	Initial draft.	4/3/03
0.70	Initial draft. Edited for formatting and grammar.	6/3/03
0.90	Incorporated industry review comments. Added new chapters for USB and SCSI. Updated the coding conventions in chapter 3. Updated for the 1.10.14.62 release of the EFI Sample Implementation. Updated the supported versions of Microsoft Visual Studio* and Windows*. Removed TBD chapters that appeared in the 0.7 version. Edited for grammar and formatting.	7/20/04
0.91	Updated for UEFI 2.0	10/31/06
0.92	New formatting	11/27/06
0.93	Review feedback incorporated	1/14/2007
0.94	Additional formatting	2/27/2007
0.95	Additional formatting	3/23/2007
0.96	Additional formatting	4/25/2008

Contents

Figures

Tables

Examples

1 Introduction

1.1 Overview

This document is designed to aid the development of UEFI drivers that follow the UEFI Driver Model that is described in the *Unified Extensible Firmware Interface Specification*, version 2.0 (hereafter referred to as the *"UEFI 2.0 Specification"*). There are several different classes of UEFI drivers and many variations of each of them. This document provides basic information for some of the most common classes of UEFI drivers. Many other driver designs are possible. In addition, this document covers the design guidelines for the different driver-related protocols, along with the design guidelines for PCI, USB, SCSI, LAN, and Graphics devices. Finally, this document discusses porting considerations for Itanium®-based platforms and EFI Byte Code (EBC) drivers and driver optimizations techniques.

1.1.1 Assumptions

This document assumes that the reader is familiar with the following:

UEFI 2.0 specification

The EFI Developer's Kit

The *EFI Developer's Kit* is also referred to as the *EDK*. There are 2 functionally equal EDKs with different build environments. This document will by default be using the *EDK*. If material pertains specifically to the *EDK Release II*, it will be specified at that point. The *EFI Developers Kit* supports at minimum the following operating systems:

Microsoft Windows XP

1.1.2 EDK Build Infrastructure

The *EDK* has a build infrastructure that supports Intel® compilers and the following versions of Microsoft Visual Studio*:

Microsoft Visual Studio .NET 2003.

This version does not require service packs.

The *EDK* Release II has a build infrastructure that supports many other compilers. Information on this can be found at www.tianocore.org.

In general, this document will use the term *Visual Studio* to refer generically to any of the supported versions in the Visual Studio tool chains.

1.2 Organization of This Document

This document is not intended to be read front to back. Instead, it is designed more as a cookbook for developing and implementing drivers. The first four chapters provide background information, chapter 5 provides the basic recipe for all drivers, and the remaining chapters provide detailed information for developing specific types of drivers.

In general, driver writers should use this document in the following way:

1. Read the first four chapters of this document before starting to code.
2. Read chapter 5, which describes the general guidelines for designing all types of UEFI drivers. Specifically, see section 5.2 for the general steps to follow when designing a driver. This section then points you to other sections in this document that contain the specific "recipe" for that particular type of UEFI driver.
3. Read the specific sections or chapters listed in section 5.2 that apply to your UEFI driver.

Table 1 describes the organization of this document.

Table 1. Organization of the UEFI 2.0 Driver Writer's Guide

Chapter		Description
1.	Introduction	Provides a brief overview of this document, its organization, and the conventions used in it.
2.	Foundation	Describes the basic concepts in the U*EFI 2.0 Specification.*
3.	Coding Conventions	Describes the common coding conventions to use in an implementation of the *UEFI 2.0 Specification.*
4.	EFI Services	Describes the services available in the *UEFI 2.0 Specification* and provides example code that uses these UEFI services.
5.	General Driver Design Guidelines	Gives general guidelines for designing all types of UEFI drivers. Provides the basic "recipe" for developing a driver and points to later chapters or sections for more detailed information.
6.	Classes of UEFI drivers	Describes the required and optional features and the subtypes for different types of drivers that follow the UEFI Driver Model.
7.	Driver Entry Point	Describes the entry point and optional features for the different classes of UEFI drivers.
8.	Private Context Data Structures	Introduces object-oriented programming techniques for managing controllers and the concept of private context data structures.
9.	Driver Binding Protocol	Describes the requirements and features for the Driver Binding Protocol. This protocol provides services that can be used to connect a driver to a controller and disconnect a driver from a controller.
10.	Component Name Protocol	Describes the requirements and features for the Component Name Protocol. This protocol provides a human-readable name for drivers and the devices that drivers manage.
11.	Driver Configuration Protocol	Describes the requirements and features for the Driver Configuration Protocol. This protocol allows the user to set the configuration options for the devices that drivers manage.

12.	Driver Diagnostics Protocol	Describes the requirements and features for the Driver Diagnostics Protocol. This protocol allows diagnostics to be executed on the devices that drivers manage.
13.	Bus Specific Driver Override Protocol	Describes the requirements and features for the Bus Specific Driver Override Protocol. This protocol matches one or more drivers to a controller. In general, this protocol applies only to bus types that provide containers for drivers on their child devices.
14.	PCI Driver Design Guidelines	Describes the guidelines that apply specifically to the management of PCI controllers.
15.	USB Driver Design Guidelines	Describes the guidelines that apply specifically to the management of USB controllers.
16.	SCSI Driver Design Guidelines	Describes the guidelines that apply specifically to the management of SCSI controllers.
17.	LAN Driver Design Guidelines	Describes the guidelines that apply specifically to the management of network controllers.
18.	Graphics Driver Design Guidelines	Describes the guidelines that apply specifically to the management of graphics controllers.
19.	Implementing the I/O Protocols	Describes how to implement I/O protocols.
20.	Driver Optimization Techniques	Describes several techniques to optimize an UEFI driver.
21.	Itanium® Architecture Porting Considerations	Describes guidelines to improve the portability of an UEFI driver and the pitfalls that may be encountered when an UEFI driver is ported to an Intel® Itanium® processor.
22.	EFI Byte Code Porting Considerations	Describes considerations when writing drivers that may be ported to EFI Byte Code (EBC).
23.	Building UEFI drivers	Describes how to write, compile, and package UEFI drivers for the *EDK* environment.
24.	Testing and Debugging UEFI drivers	Describes techniques to test and debug UEFI drivers.
25.	Distributing Device Drivers	Describes methods for distributing drivers.
26.	Utilities	Describes utilities available to help with driver development.
27.	Driver Checklist	Checklist for use in driver development.
A.	EFI Data Types	Lists the set of base data types that are used in all UEFI applications and UEFI drivers and the modifiers that can be used in conjunction with the UEFI data types.
B.	EFI Status Codes	Lists the **EFI_STATUS** code values that may be returned by EFI Boot Services, EFI Runtime Services, and UEFI protocol services.
C.	Quick Reference Guide	Provides a summary of the services, protocols, macros, constants, and GUIDs that are available to UEFI drivers.
D.	Disk I/O Protocol and Disk I/O Driver	Provides the source files to the Disk I/O Protocol, the EFI Global Variable GUID, and the source files to the disk I/O driver.

E.	Sample Drivers	Lists all the sample drivers that are available in the *EDK*
F.	Glossary	Lists and defines the terms and acronyms used in this document.

1.3 Related Information

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

1.3.1 EFI Specifications

Unified Extensible Firmware Interface, version 2.0, The UEFI Forum, 2006, http://www.uefi.org

Unified Extensible Firmware Interface, version 2.1, The UEFI Forum, 2007, http://www.uefi.org

Platform Initialization Architecture, version 1.0, The UEFI Forum, 2006, http://www.uefi.org

1.3.2 Pci Specifications

Conventional PCI specification 2.3, *Conventional PCI Specification 3.0*, *PCI Express Base Specification 1.1*, and *PCI-X Protocol Specification 2.0a*, *PciSig*, http://www.pcisig.com *(Collectively referred to as PCI Specification)*

This applies to future versions of the aforementioned PCI specifications.

1.3.3 USB Specifications

Universal Serial Bus Revision 2.0 specification bundle, USB Implementers Forum, Inc., 2006, http://www.usb.org (this bundle is referred to as USB Spec).

This applies to future versions of the aforementioned USB specifications.

1.3.4 Graphics Specifications

E-DID EEPROM Specification, VESA, http://www.vesa.org

1.3.5 Other Specifications

UEFI driver Library Specification, Intel Corporation, http://www.tianocore.org

Glue Library Programming Guid and *Glue Library Reference Manual version 0.58* (or higher), Intel Corporation 2007, http://www.tianocore.org

Itanium® Processor Family System Abstraction Layer Specification, Intel Corporation,

Intel® Itanium® Architecture Software Developer's Manual, vols. 1–4, Intel Corporation,

A Formal Specification of Intel® Itanium® Processor Family Memory Ordering, Intel Corporation, http://developer.intel.com/design/itanium/family

Microsoft Portable Executable and Common Object File Format Specification, Microsoft Corporation, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

Developer's Interface Guide for 64-bit Intel Architecture-based Servers, version. 2.1, Compaq Computer Corporation, Dell Computer Corporation, Fujitsu Siemens Computers, Hewlett-Packard Company, Intel Corporation, International Business Machines Corporation, and NEC Corporation, 2001, http://www.dig64.org/specifications

1.3.6 Books

Code Complete, Steven C. McConnell, ISBN 1-55615-484-4

Beyond Bios: Implementing the Unified Extensible Firmware Interface with Intel's Framework, Vincent Zimmer, Michael Rothman, and Robert Hale, ISBN 0-9743649-0-8, http://www.intel.com/intelpress/sum_efi.htm

1.3.7 Tools

EFI Developer's Kit and *EFI Developer's Kit* Release II, Intel Corporation, 2007, http://www.tianocore.org (known hereafter as EDK and EDK II respectively)

EFI Shell, *UEFI Shell*, and *EFI Shell Users Guide*, Intel Corporation, http://www.tianocore.org

1.4 Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

1.4.1 Data Structure Descriptions

Intel® processors based on 32-bit Intel® Instruction Set Architecture (IA-32) are "little endian" machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the Itanium processor 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.

1.4.2 Pseudo-Code Conventions

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

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

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

1.4.3 Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text The normal text typeface is used for the vast majority of the descriptive text in a specification.

Plain text (blue) In the electronic version of this specification, any plain text underlined and in blue indicates an active link to the cross-reference.

Bold In text, a *Bold* typeface identifies a processor register name. In other instances, a *Bold* typeface can be used as a running head within a paragraph.

Italic In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.

BOLD Monospace Computer code, example code segments, and all prototype code segments use a **BOLD Monospace** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.

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

Plain Monospace In code, words in a **Plain Monospace** typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs and can be outlined by a thin black border.

2 Foundation

There are several UEFI concepts that are cornerstones for understanding UEFI drivers. These concepts are defined in the *UEFI 2.0 Specification*. However, programmers who are new to UEFI may find the following introduction to a few of the key UEFI concepts a helpful framework to keep in mind as they study the *UEFI 2.0 Specification*.

The basic concepts that are covered in the following sections include the following:

- Objects managed by UEFI-compliant firmware
- EFI System Table
- Handle database
- Protocols
- UEFI images
- Events
- Task Priority Levels
- Device paths
- UEFI Driver Model
- Platform initialization
- Boot manager and console management

As each concept is discussed, the related application programming interfaces (APIs) will be identified along with references to the related sections in the *UEFI 2.0 Specification*. One component that is distributed with the *EFI Developer's Kit* is the UEFI Shell. The UEFI Shell is an UEFI application that provides the user with a command line interface. This command line interface provides commands that are useful in the development and testing of UEFI drivers and UEFI applications. In addition, the UEFI Shell provides commands that can help illustrate many of the basic concepts that are described in the following sections. The useful UEFI Shell commands will be identified as each concept is introduced. The UEFI Shell is an open source project at www.tianocore.org and has documents that detail all the available commands whether mentioned here or not.

2.1 Objects Managed by EFI-Based Firmware

There are several different types of objects that can be managed through the services provided by EFI. Figure 1 below shows the various object types. The most important ones for UEFI drivers are the following:

- EFI System Table
- Memory

- Handles
- Images
- Events

Some UEFI drivers may need to access environment variables, but most do not. Rarely do UEFI drivers require the use of a monotonic counter, watchdog timer, or real-time clock. The EFI System Table provides access to all the services provided by UEFI and access to all the additional data structures that describe the configuration of the platform. Each of these object types and the services that provide access to them will be introduced in the following sections.

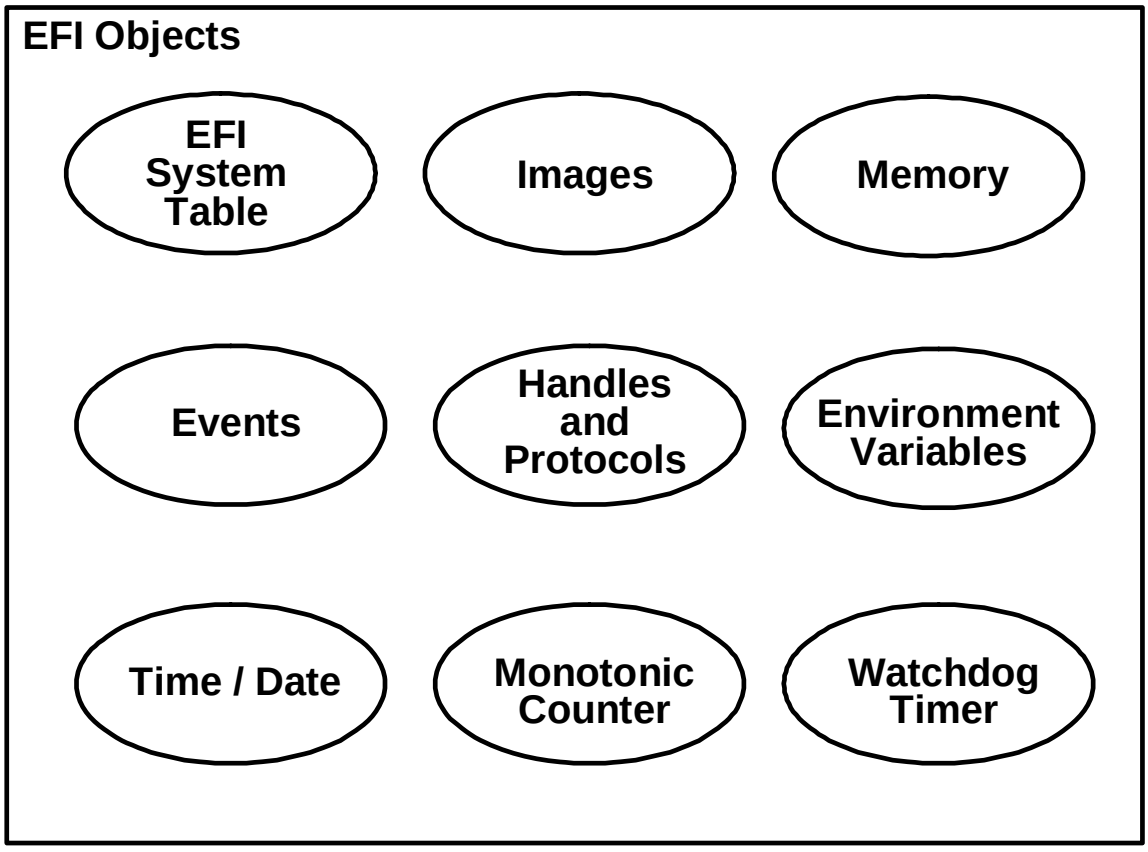

Figure 1. Object Managed by UEFI-Based Firmware

2.2 EFI System Table

The *EFI System Table* is the most important data structure in UEFI. From this one data structure, an UEFI executable image can gain access to system configuration information and a rich collection of UEFI services. These UEFI services include the following:

- EFI Boot Services
- EFI Runtime Services
- Protocol services

The EFI Boot Services and EFI Runtime Services are accessed through the EFI Boot Services Table and the EFI Runtime Services Table respectively, which are two of the data fields in the EFI System Table. The number and type of services that are available from these two tables is fixed for each revision of the *UEFI Specification*. The EFI Boot Services and EFI Runtime Services are defined in the *UEFI 2.0 Specification*, and the common uses of theses services by UEFI drivers are presented in chapters 6 and 7 of the *UEFI 2.0 Specification.*

Protocol services are groups of related functions and data fields that are named by a Globally Unique Identifier (GUID; see Appendix A of the *UEFI 2.0 Specification*). Protocol services are typically used to provide software abstractions for devices such as consoles, disks, and networks. They can also be used to extend the number of generic services that are available in the platform. Protocols are the basic building blocks that allow the functionality of UEFI firmware to be extended over time. The *UEFI 2.0 Specification* defines over 30 different protocols, and various implementations of UEFI firmware and UEFI drivers may produce additional protocols to extend the functionality of a platform.

2.3 Handle Database

The *handle database* is composed of objects called handles and protocols. *Handles* are a collection of one or more protocols, and *protocols* are data structures that are named by a GUID. The data structure for a protocol may be empty, may contain data fields, may contain services, or may contain both services and data fields. During platform initialization, the system firmware, UEFI compliant drivers, and UEFI applications will create handles and attach one or more protocols to the handles. Information in the handle database is "global" and can be accessed by any executable UEFI image.

The handle database is the central repository for the objects that are maintained by UEFI-based firmware. The handle database is a list of UEFI handles. Each UEFI handle is identified by a unique handle number that is maintained by the system firmware. A handle number provides a database "key" to an entry in the handle database. Each entry in the handle database is a collection of one or more protocols. The types of protocols, named by GUID, that are attached to an UEFI handle determine the handle type. An UEFI handle may represent components such as the following:

- Executable images such as UEFI drivers and UEFI applications
- Devices such as network controllers and hard drive partitions
- UEFI service (drivers) such as EFI Decompress and the EBC Interpreter

Figure 2 below shows a portion of the handle database. In addition to the handles and protocols, a list of objects is associated with each protocol. This list is used to track which agents are consuming which protocols. This information is critical to the operation of UEFI drivers, because this information is what allows UEFI drivers to be safely loaded, started, stopped, and unloaded without any resource conflicts.

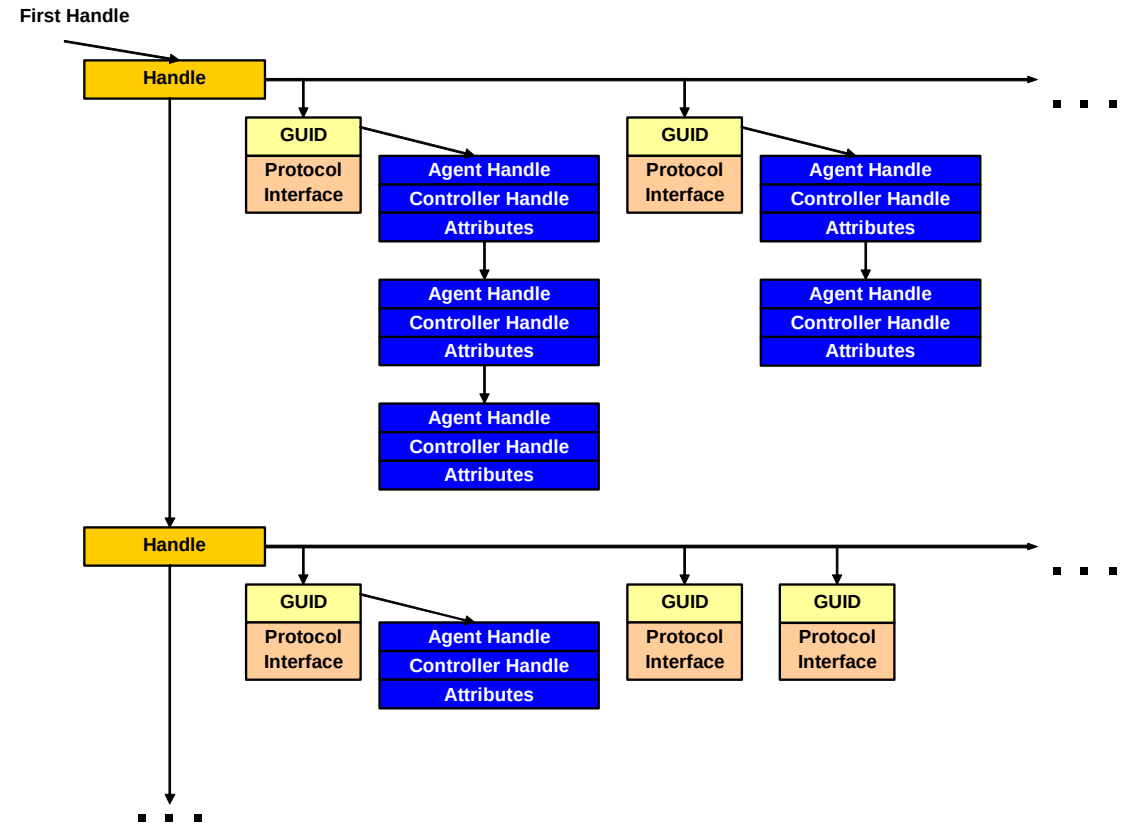

Figure 2. Handle Database

Figure 3 below shows the different types of handles that may be present in the handle database and the relationships between the various handle types. The handle-related terms introduced here are used throughout the document. All handles reside in the same handle database and the types of protocols that are associated with each handle differentiate the handle type.

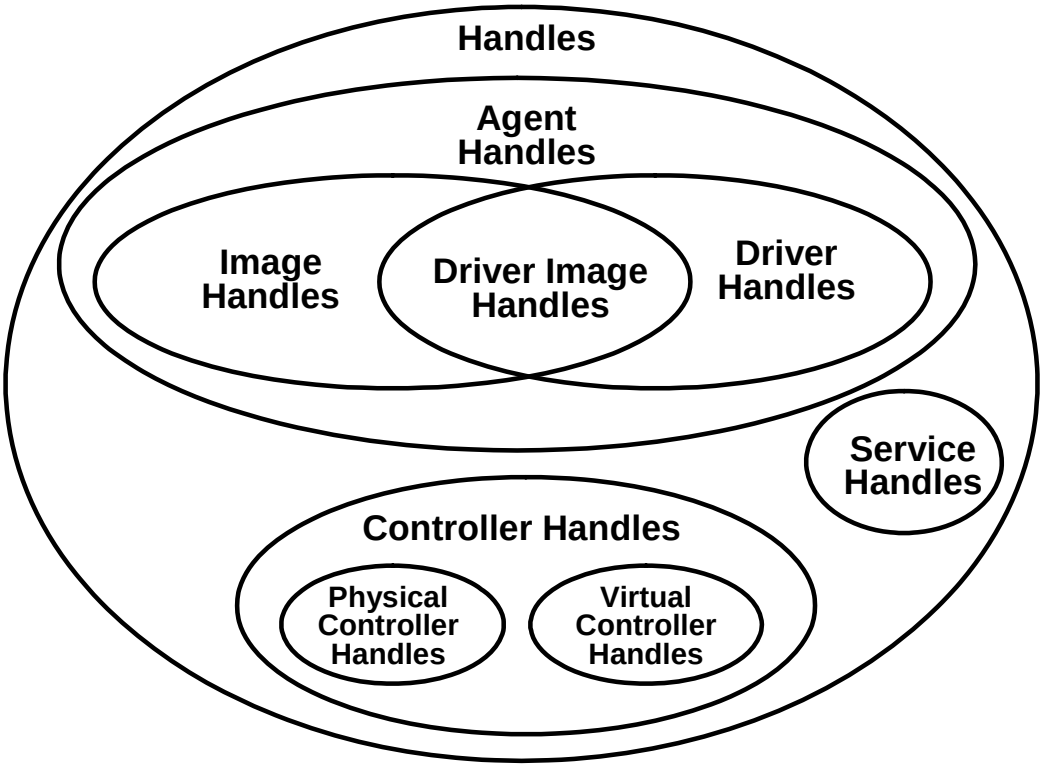

Figure 3. Handle Types

Table 2. describes the types of handles that are shown in Figure 3.

Table 2. Description of Handle Types

Type of Handle	Description
Agent handle	This term is used by some of the UEFI Driver Model–related services in the *UEFI 2.0 Specification*. An *agent* is a UEFI component that can consume a protocol in the handle database. An *agent handle* is a general term that can represent an image handle, a driver handle, or a driver image handle.
Image handle	Supports the EFI Loaded Image Protocol. See chapter 8 of the *UEFI 2.0 Specification*.
Driver handle	Supports the Driver Binding Protocol. May optionally support the Driver Configuration Protocol, the Driver Diagnostics Protocol, and the Component Name Protocol. *DIG64* requires these optional protocols for Itanium-based platforms. Other platform specifications may or may not require these protocols. See chapter 10 of the *UEFI 2.0 Specification*.

Driver image handle	The intersection of image handles and driver handles. Supports both the Loaded Image Protocol and the Driver Binding Protocol. May optionally support the Driver Configuration Protocol, the Driver Diagnostics Protocol, and the Component Name Protocol. *DIG64* requires these optional protocols for Itanium-based platforms. Other platform specifications may or may not require these protocols. See chapter 10 of the *UEFI 2.0 Specification*.
Controller handle	If the handle represents a physical device, then it must support the Device Path Protocol. If the handle represents a virtual device, then it must not support the Device Path Protocol. In addition, a device handle must support one or more additional I/O protocols that are used to abstract access to that device. The list of I/O protocols that are defined in the *UEFI 2.0 Specification* include the following: Console Services: Simple Input Protocol, Simple Text Output Protocol, Simple Pointer Protocol, Serial I/O Protocol, and Debug Port Protocol Bootable Image Services: Block I/O Protocol, Disk I/O Protocol, Simple File System Protocol, and Load File Protocol Network Services: Network Interface Identifier Protocol, Simple Network Protocol, and PXE Base Code Protocol PCI Services: PCI Root Bridge I/O Protocol, and PCI I/O Protocol USB Services: USB Host Controller Protocol and USB I/O Protocol SCSI Services: Ext SCSI Pass Thru Protocol and SCSI I/O Protocol Graphics Services: Graphics Output Protocol
Device handle	Term is used interchangeably with *controller handle*.
Bus controller handle	Managed by a bus driver or a hybrid driver that produces child handles. The term "bus" does not necessarily match the hardware topology. The term "bus" in this document is used from the software perspective and the production of the software construct—which is called a child handle—is the only distinction between a controller handle and a bus controller handle.
Child handle	A type of controller handle that is created by a bus driver or a hybrid driver. See chapter 6 (sections 6.2 and 6.3) for the definitions of bus hybrid drivers. The distinction between a child handle and a controller handle depends on the perspective of the driver that is using the handle. A handle would be a child handle from a bus driver's perspective, and that same handle may be a controller handle from a device driver's perspective.
Physical controller handle	A controller handle that represents a physical device that must support the Device Path Protocol. See chapter 9 of the *UEFI 2.0 Specification.*
Virtual controller handle	A controller handle that represents a virtual device and does not support the Device Path Protocol.
Service handle	Does not support the Loaded Image Protocol, the Driver Binding Protocol, or the Device Path Protocol. Instead, it supports the only instance of a specific protocol in the entire handle database. This protocol provides services that may be used by other UEFI applications or UEFI drivers. The list of service protocols that are defined in the *UEFI 2.0 Specification* include the Platform Driver Override Protocol, Unicode Collation Protocol, Boot Integrity Services Protocol, Debug Support Protocol, Decompress Protocol, and EFI Byte Code Protocol.

2.4 Protocols

The extensible nature of UEFI is built, to a large degree, around protocols. UEFI drivers are sometimes confused with UEFI protocols. Although they are closely related, they are distinctly different. An UEFI driver is an executable UEFI image that installs a variety of protocols of various handles to accomplish its job.

UEFI protocols are a block of function pointers and data structures or APIs that have been defined by a specification. At a minimum, the specification will define a GUID. This number is the protocol's real name and will be used to find this protocol in the handle database. The protocol also typically includes a set of procedures and/or data structures (called the *protocol interface structure*). The following is an example of a protocol definition from section 10.6 of the *UEFI 2.0 Specification*. Notice that it defines two function definitions and one data field.

GUID

```
#define EFI_COMPONENT_NAME_PROTOCOL_GUID \
  { 0x107a772c,0xd5e1,0x11d4,0x9a,0x46,0x0,0x90,0x27,0x3f,0xc1,0x4d }
```

Protocol Interface Structure

```
typedef struct _EFI_COMPONENT_NAME_PROTOCOL {
  EFI_COMPONENT_NAME_GET_DRIVER_NAME      GetDriverName;
  EFI_COMPONENT_NAME_GET_CONTROLLER_NAME  GetControllerName;
  CHAR8                                   *SupportedLanguages;
} EFI_COMPONENT_NAME_PROTOCOL;
```

Figure 4 below shows a single handle and protocol from the handle database that is produced by an UEFI driver. The protocol is composed of a GUID and a protocol interface structure. Many times, the UEFI driver that produces a protocol interface will maintain additional private data fields. The protocol interface structure itself simply contains pointers to the protocol function. The protocol functions are actually contained within the UEFI driver. An UEFI driver may produce one protocol or many protocols depending on the driver's complexity.

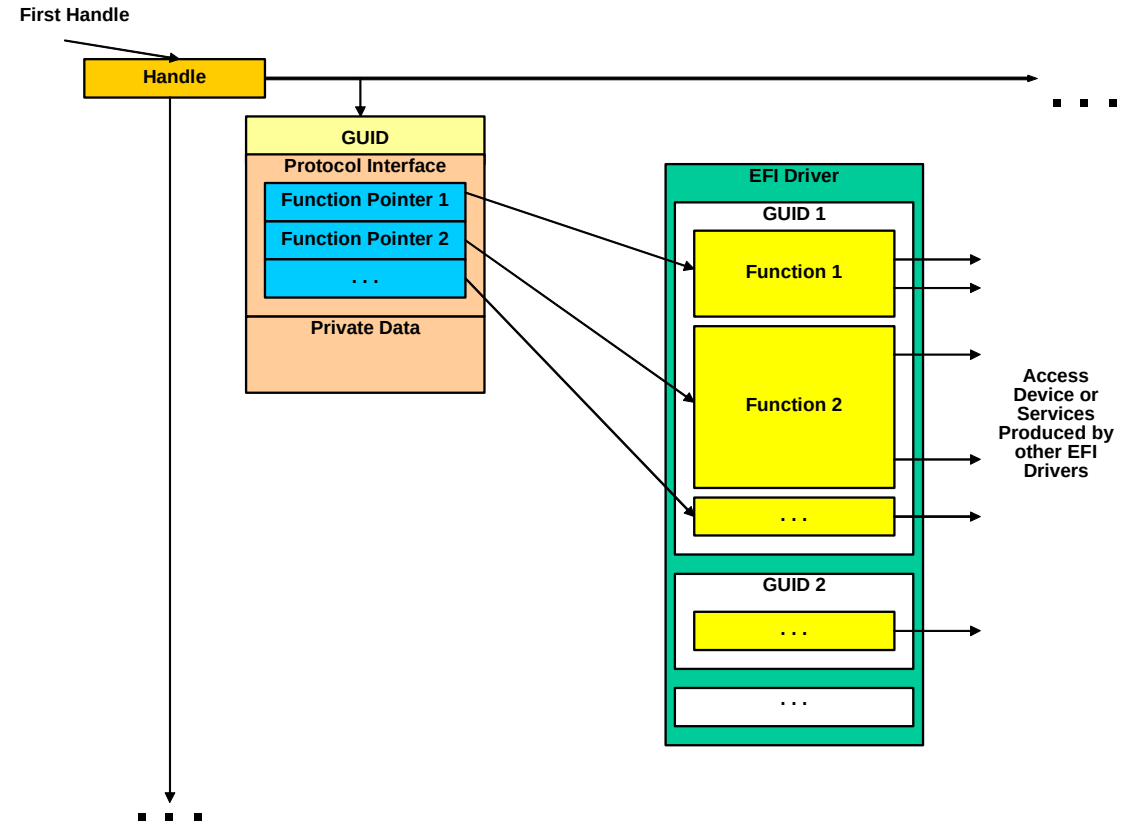

Figure 4. Construction of a Protocol

Not all protocols are defined in the *UEFI 2.0 Specification*. The *EFI Developer's Kit* includes some protocols that are not part of the *UEFI 2.0 Specification*. These protocols are necessary to provide all of the functionality in a particular implementation, but they are not defined in the *UEFI 2.0 Specification* because they do not present an external interface that is required to support booting an OS or writing an UEFI driver. The creation of new protocols is how UEFI-based systems can be extended over time as new devices, buses, and technologies are introduced.

The following are a few examples of protocols in the *EFI Developer's Kit* that are not part of the *UEFI 2.0 Specification:*

- *Varstore*
- *ConIn*
- *ConOut*
- *StdErr*
- *PrimaryConIn*
- *VgaMiniPort*

- *UsbAtapi*

The UEFI application Toolkit also contains a number of UEFI protocols, such as the following, that may be found on some platforms:

- PPP Daemon
- Ramdisk

The OS loader and drivers should not depend on these types of protocols because they are not guaranteed to be present in every UEFI-compliant firmware implementation. OS loaders and drivers should depend only on protocols that are defined in the *UEFI 2.0 Specification* and protocols that are required by platform design guides such as *DIG64*.

The extensible nature of UEFI allows each platform to design and add its own special protocols. These protocols can be used to expand that capabilities of UEFI and provide access to proprietary devices and interfaces in congruity with the rest of the UEFI architecture.

Because a protocol is "named" by a GUID, there should be no other protocols with that same identification number. Care must be taken when creating a new protocol to define a new GUID for it. UEFI fundamentally assumes that a specific GUID will expose a specific protocol interface. Cutting and pasting an existing GUID or hand modifying an existing GUID creates the opportunity for a duplicate GUID to be introduced. A system containing a duplicate GUID may inadvertently find the new protocol and think that it is another protocol, which will mostly likely crash the system. These types of bugs are also very difficult to root cause. The versions of Microsoft Visual Studio that are referenced by this document include a **`GUIDGEN`** tool that can quickly generate a GUID definition that is suitable for including in the code.

2.4.1 Working with Protocols

Any UEFI image can operate with protocols during boot time. However, after **`ExitBootServices()`** is called, the handle database is no longer available. There are several UEFI boot time services for working with UEFI protocols. Section 6.3 of the *UEFI 2.0 Specification* discusses these Protocol Handler Services. These services are described in more detail in chapter 4 of this document.

2.4.2 Multiple Protocol Instances

A handle may have many protocols attached to it. However, it may have only one protocol of each type. In other words, a single handle may not produce more than one instance of any single protocol. This prevents nondeterministic behavior about which instance would be consumed by a given request.

However, drivers may create multiple "instances" of a particular protocol and attach each instance to a different handle. This scenario is the case with the PCI I/O Protocol, where the PCI bus driver installs a PCI I/O Protocol instance for each PCI device. Each "instance" of the PCI I/O Protocol is configured with data values that are unique to that PCI device, including the location and size of the UEFI compliant Option ROM (OpROM) image.

Each driver can install customized versions of the same protocol (as long as it is not on the same handle). For example, each UEFI driver produces the Component Name Protocol on its driver image handle, yet when the **EFI_COMPONENT_NAME_PROTOCOL.GetDriverName()** function is called each handle will return the unique name of the driver that owns that image handle. The **EFI_COMPONENT_NAME_PROTOCOL.GetDriverName()** function on the USB bus driver handle will return "USB bus driver" for the English language, but the **EFI_COMPONENT_NAME_PROTOCOL.GetDriverName()** function on the PXE driver handle will return "PXE base code driver."

2.4.3 Tag GUID

A protocol may be nothing more than a GUID. This GUID is also known as a *tag GUID*. Such a protocol can still serve very useful purposes such as marking a device handle as special in some way or allowing other UEFI images to find the device handle easily by querying the system for the device handles with that protocol GUID attached.

The *EFI Developers Kit* uses the **HOT_PLUG_DEVICE_GUID** in this way to mark device handles that represent devices from a hot-pluggable bus (such as USB).

2.5 UEFI images

This section describes the different types of UEFI images. All UEFI images contain a PE/COFF header that defines the format of the executable code (see the *Microsoft Portable Executable and Common Object File Format Specification for more information*). The code can be for IA-32, IA-64, EBC, or x64. The header will define the processor type and the image type. Section 2.1.1 of the *UEFI 2.0 Specification* defines the processor types and the following three image types:

- EFI applications
- EFI Boot Services drivers
- EFI Runtime drivers

UEFI images are loaded and relocated into memory with the Boot Service **gBS->LoadImage()**. There are several supported storage locations for UEFI images, including the following:

- Expansion ROMs on a PCI card
- System ROM or system flash
- A media device such as a hard disk, floppy, CD-ROM, or DVD
- LAN server

In general, UEFI images are not compiled and linked at a specific address. Instead, they are compiled and linked such that relocation fix-ups are included in the UEFI image, which allows the UEFI image to be placed anywhere in system memory. The Boot Service **gBS->LoadImage()** does the following:

- Allocates memory for the image being loaded
- Automatically applies the relocation fix-ups to the image

- Creates a new image handle in the handle database, which installs an instance of the **EFI_LOADED_IMAGE_PROTOCOL**

This instance of the **EFI_LOADED_IMAGE_PROTOCOL** contains information about the UEFI image that was loaded. Because this information is published in the handle database, it is available to all UEFI components.

After an UEFI image is loaded with **gBS->LoadImage()**, it can be started with a call to **gBS->StartImage()**. The header for an UEFI image contains the address of the entry point that is called by **gBS->StartImage()**. The entry point always receives the following two parameters:

- The image handle of the UEFI image being started
- A pointer to the EFI System Table

These two items allow the UEFI image to do the following:

- Access all of the UEFI services that are available in the platform.
- Retrieve information about where the UEFI image was loaded from and where in memory the image was placed.

The operations that are performed by the UEFI image in its entry point vary depending on the type of UEFI image. The figure below shows the various UEFI image types and the relationships between the different levels of images.

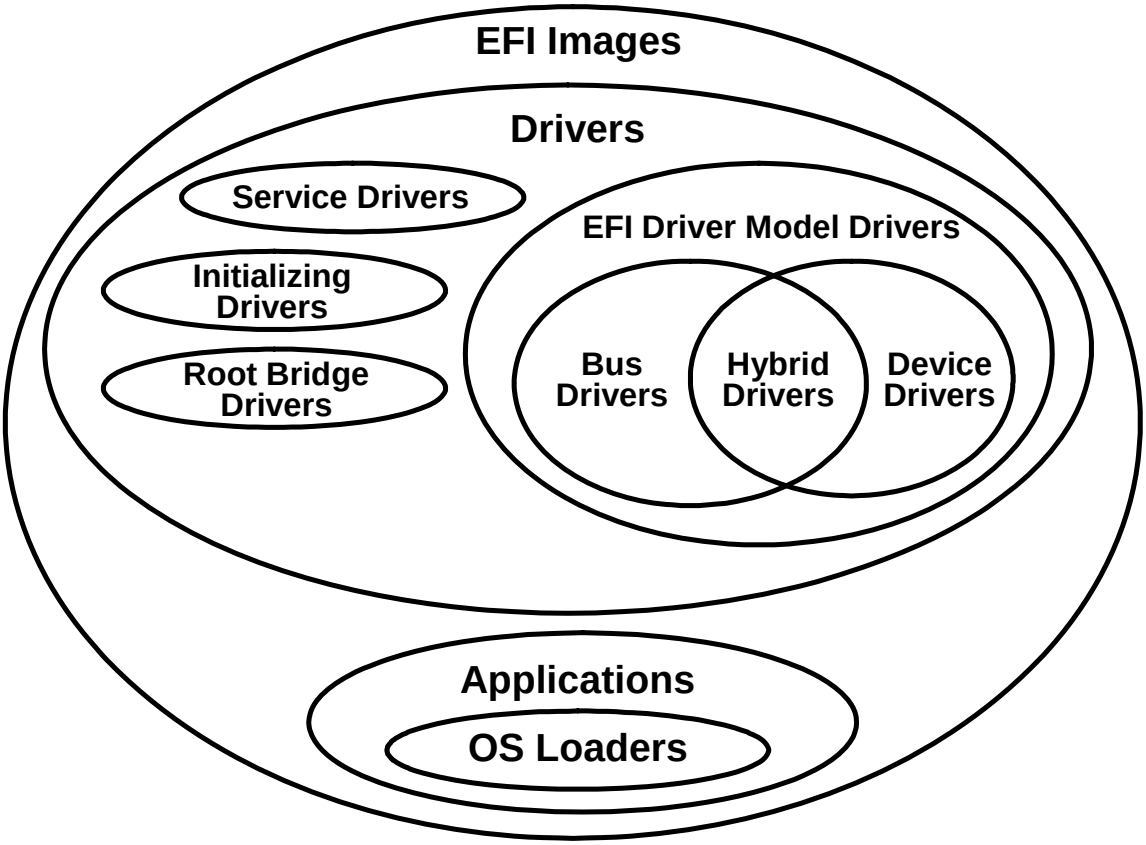

Figure 5. Image Types

The table below describes the types of images that were shown in the preceding figure.

Table 3. Description of Image Types

Type of Image	Description
Application	An UEFI image of type **`EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION`**. This image is executed and automatically unloaded when the image exits or returns from its entry point.
OS loader	A special type of application that normally does not return or exit. Instead, it calls the EFI Boot Service **`gBS->ExitBootServices()`** to transfer control of the platform from the firmware to an operating system.

Driver	An UEFI image of type **EFI_IMAGE_SUBSYSTEM_BOOT_SERVICE_DRIVER** or **EFI_IMAGE_SUBSYSTEM_RUNTIME_DRIVER**. If this image returns **EFI_SUCCESS**, then the image is not unloaded. If the image returns an error code other than **EFI_SUCCESS**, then the image is automatically unloaded from system memory. The ability to stay resident in system memory is what differentiates a driver from an application. Because drivers can stay resident in memory, they can provide services to other drivers, applications, or an operating system. Only the services produced by runtime drivers are allowed to persist past **gBS->ExitBootServices()**.
Service driver	A driver that produces one or more protocols on one or more new service handles and returns **EFI_SUCESS** from its entry point.
Initializing driver	A driver that does not create any handles and does not add any protocols to the handle database. Instead, this type of driver performs some initialization operations and returns an error code so the driver is unloaded from system memory.
Root bridge driver	A driver that creates one or physical controller handles that contain a Device Path Protocol and a protocol that is a software abstraction for the I/O services provided by a root bus produced by a core chipset. The most common root bridge driver is one that creates handles for the PCI root bridges in the platform that support the Device Path Protocol and the PCI Root Bridge I/O Protocol.
UEFI Driver Model driver	A driver that follows the UEFI Driver Model that is described in detail in the *UEFI 2.0 Specification*. This type of driver is fundamentally different from service drivers, initializing drivers, and root bridge drivers because a driver that follows the UEFI Driver Model is not allowed to touch hardware or produce device-related services in the driver entry point. Instead, the driver entry point of a driver that follows the UEFI Driver Model is allowed only to register a set of services that allow the driver to be started and stopped at a later point in the system initialization process.
Device driver	A driver that follows the UEFI Driver Model. This type of driver produces one or more driver handles or driver image handles by installing one or more instances of the Driver Binding Protocol into the handle database. This type of driver does not create any child handles when the **Start()** service of the Driver Binding Protocol is called. Instead, it only adds additional I/O protocols to existing controller handles.
Bus driver	A driver that follows the UEFI Driver Model. This type of driver produces one or more driver handles or driver image handles by installing one or more instances of the Driver Binding Protocol in the handle database. This type of driver creates new child handles when the **Start()** service of the Driver Binding Protocol is called. It also adds I/O protocols to these newly created child handles.
Hybrid driver	A driver that follows the UEFI Driver Model and shares characteristics with both device drivers and bus drivers. This distinction means that the **Start()** service of the Driver Binding Protocol will add I/O protocols to existing handles and it will create child handles.

2.5.1 Applications

An UEFI application will start execution at its entry point and then execute until it returns from its entry point or it calls the **Exit()** boot service function. When it is done,

the image is unloaded from memory. It does not stay resident in memory. Some examples of common UEFI applications include the following:

- EFI Shell
- EFI Shell commands
- Flash utilities
- Diagnostic utilities

It is perfectly acceptable to invoke UEFI applications from inside other UEFI applications.

2.5.1.1 OS Loader

The *UEFI 2.0 Specification* details a special type of UEFI application called an *OS boot loader.* It is an UEFI application that calls **`ExitBootServices()`**. **`ExitBootServices()`** is called when the OS loader has set up enough of the OS infrastructure that it is ready to assume ownership of the system resources. At **`ExitBootServices()`**, the UEFI core will free all of its boot time services and drivers, leaving only the runtime services and drivers.

2.5.2 Drivers

UEFI drivers are different from UEFI applications in that unless there is an error returned from the driver's entry point, the driver stays resident in memory. The UEFI core firmware, the boot manager, or other UEFI applications may load drivers.

2.5.2.1 Boot Service Drivers

Boot time drivers are loaded into memory marked as **`EfiBootServicesCode`**, and they allocate their data structures from memory marked as **`EfiBootServicesData`**. These memory types are converted to available memory after **`gBS->ExitBootServices()`** is called.

2.5.2.2 Runtime Drivers

Runtime drivers are loaded in memory marked as **`EfiRuntimeServicesCode`**, and they allocate their data structures from memory marked as **`EfiRuntimeServicesData`**. These types of memory are preserved after **`gBS->ExitBootServices()`** is called. This preservation allows runtime driver to provide services to an operating system while the operating system is running. Runtime drivers must publish an alternative calling mechanism, because the UEFI handle database does not persist into OS runtime. The most common examples of UEFI runtime drivers are the Floating Point Software Assist driver (**`FPSWA.efi`**) and the network Universal Network Driver Interface (UNDI) driver. Other than these examples, runtime drivers are not very common and will not be discussed in detail. In addition, the implementation and validation of runtime drivers is much more difficult than the implementation and validation of boot service drivers because UEFI supports the translation of runtime services and runtime drivers from a physical addressing mode to a virtual addressing mode.

2.6 Events and Task Priority Levels

Events are another type of object that is managed through UEFI services. They can be created and destroyed and are either in the waiting state or the signaled state. An UEFI image can do any of the following:

- Create an event.
- Destroy an event.
- Check to see if an event is in the signaled state.
- Wait for an event to be in the signaled state.
- Request that an event be moved from the waiting state to the signaled state.

Because UEFI does not support interrupts, it can present a challenge to driver writers who are accustomed to an interrupt-driven driver model. Instead, UEFI supports polled drivers. The most common use of events by an UEFI driver is the use of timer events that allows drivers to poll a device periodically. . Figure 6 below shows the different types of events that are supported in UEFI and the relationships between those events.

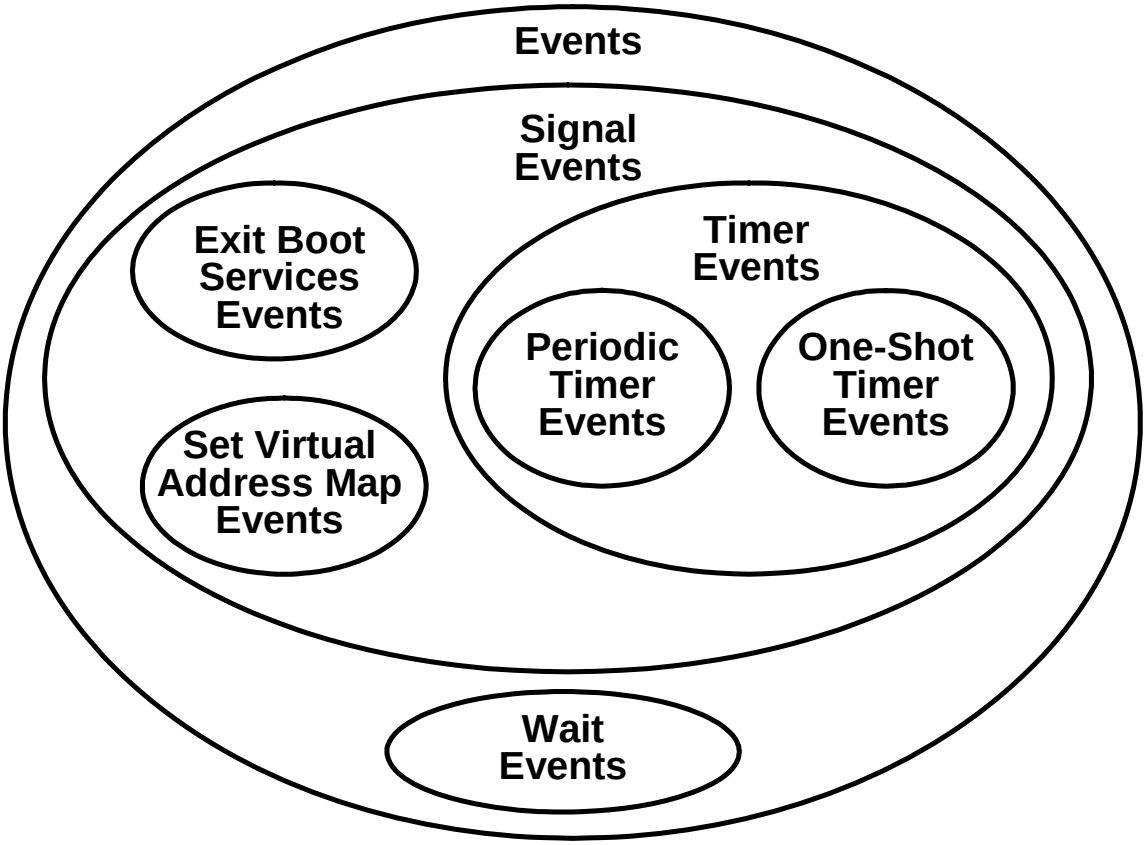

Figure 6. Event Types

The following table describes the types of events that were shown the preceding figure.

Table 4. Description of Event Types

Type of Events	Description
Wait event	An event whose notification function is executed whenever the event is checked or waited upon.
Signal event	An event whose notification function is scheduled for execution whenever the event goes from the waiting state to the signaled state.
Exit Boot Services event	A special type of signal event that is moved from the waiting state to the signaled state when the EFI Boot Service **`ExitBootServices()`** is called. This call is the point in time when ownership of the platform is transferred from the firmware to an operating system. The event's notification function is scheduled for execution when **`ExitBootServices()`** is called.

Set Virtual Address Map event	A special type of signal event that is moved from the waiting state to the signaled state when the EFI Runtime Service **SetVirtualAddressMap()** is called. This call is the point in time when the operating system is making a request for the runtime components of UEFI to be converted from a physical addressing mode to a virtual addressing mode. The operating system provides the map of virtual addresses to use. The event's notification function is scheduled for execution when **SetVirtualAddressMap()** is called.
Timer event	A type of signal event that is moved from the waiting state to the signaled state when at least a specified amount of time has elapsed. Both periodic and one-shot timers are supported. The event's notification function is scheduled for execution when a specific amount of time has elapsed.
Periodic timer event	A type of timer event that is moved from the waiting state to the signaled state at a specified frequency. The event's notification function is scheduled for execution when a specific amount of time has elapsed.
One-shot timer event	A type of timer event that is moved from the waiting state to the signaled state after the specified timer period has elapsed. The event's notification function is scheduled for execution when a specific amount of time has elapsed.

The following three elements are associated with every event:

- The Task Priority Level (TPL) of the event
- A notification function
- A notification context

The notification function for a wait event is executed when the state of the event is checked, or when the event is being waited upon. The notification function of a signal event is executed whenever the event transitions from the waiting state to the signaled state. The notification context is passed into the notification function each time the notification function is executed. The TPL is the priority at which the notification function is executed. The four TPL levels that are defined in EFI are listed below.

Table 5. Task Priority Levels Defined in EFI

Task Priority Level	Description
TPL_APPLICATION	The priority level at which UEFI images are executed.
TPL_CALLBACK	The priority level for most notification functions.
TPL_NOTIFY	The priority level at which most I/O operations are performed.
TPL_HIGH_LEVEL	The priority level for the one timer interrupt supported in EFI. (Not usable by Drivers)

TPLs serve the following two purposes:

- To define the priority in which notification functions are executed
- To create locks

In the first purpose, this mechanism is used only when more than one event is in the signaled state at the same time. In these cases, the notification function that has been registered with the higher priority will be executed first. Also, notification functions at higher priorities can interrupt the execution of notification functions executing at a lower priority.

In the second purpose, creating locks, it is possible for the code in normal context and the code in interrupt context to access the same data structure, because UEFI does support a single timer interrupt. This access can cause issues if the updates to a shared data structure are not atomic. An UEFI application or UEFI driver that wants to guarantee exclusive access to a shared data structure can temporarily raise the task priority level to prevent simultaneous access from both normal context and interrupt context. A lock can be created by temporarily raising the task priority level to **TPL_HIGH_LEVEL**. This level will even block the one timer interrupt, but care must be taken to minimize the amount of time that the system is at **TPL_HIGH_LEVEL**, because all timer-based events are blocked during this time and any driver that requires periodic access to a device will be prevented from accessing its device.

2.7 EFI Device Paths

EFI defines a Device Path Protocol that is attached to device handles in the handle database to help operating systems and their loaders identify the hardware that a device handle represents. The Device Path Protocol provides a unique name for each physical device in a system. The collection of Device Path Protocols for the physical devices managed by EFI-based firmware is called a name space. Modern operating systems tend to use ACPI and industry standard buses to produce a name space while the operating system is running. However, the ACPI name space is difficult to parse, and it would greatly increase the size and complexity of system firmware to carry an ACPI name space parser. Instead, UEFI uses aspects of the ACPI name space that do not require an ACPI name space parser. This compromise keeps the size and complexity of system firmware to a minimum and provides a way for the operating system to create a mapping from UEFI device paths to the operating system's name space.

A *device path* is a data structure that is composed of one or more device path nodes. Every device path node contains a standard header that includes the node's type, subtype, and length. This standard header allows a parser of a device path to hop from one node to the next without having to understand every type of node that may be present in the system. Example 1 below shows the declaration of the standard header for an UEFI device path. The device or bus-specific node data follows the *Length* field. The declaration of the PCI device path node is also shown in the example below. It contains the PCI-device-specific fields *Function* and *Device*.

```
//
// Generic device path node header
//
typedef struct
  UINT8  Type;
  UINT8  SubType;
  UINT8  Length[2];
} EFI_DEVICE_PATH_PROTOCOL;

//
// PCI Device Path Node that includes a header and PCI-specific fields
//
typedef struct _PCI_DEVICE_PATH {
  EFI_DEVICE_PATH_PROTOCOL        Header;
  UINT8                           Function;
  UINT8                           Device;
} PCI_DEVICE_PATH;
```

Example 1. EFI Device Path Header

A device path is position independent because it is not allowed to contain any pointer values. This independence allows device paths to be easily moved from one location to another and stored in nonvolatile storage. A device path is terminated by a special device path node called an *end device path node*. Table 6 lists the types of device path nodes that are defined in section 8.3 of the *UEFI 2.0 Specification.*

Table 6. Types of Device Path Nodes Defined in UEFI 2.0 Specification

Type of Device Path Nodes	Description
Hardware device path node	Used to describe devices on industry-standard buses that are directly accessible through processor memory or processor I/O cycles. These devices include memory-mapped devices and devices on PCI buses and PC card buses.
ACPI device path node	Used to describe devices whose enumeration is not described in an industry-standard fashion. This type of device path is used to describe devices such as PCI root bridges and ISA devices. These device path nodes contain HID, CID, and UID fields that must match the HID, CID, and UID values that are present in the platform's ACPI tables.
Messaging device path node	Used to describe devices on industry-standard buses that are not directly accessible through processor memory or processor I/O cycles. These devices are accessed by the processor through one or more hardware bridge devices that translate one industry-standard bus type to another industry-standard bus type. This type of device path is used to describe devices such as SCSI, Fibre Channel, 1394, USB, I2O, InfiniBand*, UARTs, and network agents.
Media device path node	Hard disk, CD-ROM, and file paths in a file system that support multiple directory levels.

BIOS Boot Specification device path node	Used to describe a device that has a type that follows the *BIOS Boot Specification*, such as floppies, hard disks, CD-ROMs, PCMCIA devices, USB devices, network devices, and Bootstrap Entry Vector (BEV) devices. These device path nodes are used only in a platform that supports BIOS INT services.
End device path node	Used to terminate a device path.

Each of the device path node types also supports a vendor-defined node that is the extensibility mechanism for device paths. As new devices, bus types, and technologies are introduced into platforms, new device path nodes may have to be created. The vendor-defined nodes use a GUID to distinguish new device path nodes. When a device path node is created for a new device or bus type, the **GUIDGEN** tool should be used to create a new GUID. Care must be taken in the choice of the data fields used in the definition of a new device path node. As long as a device is not physically moved from one location in a platform to another location, the device path must not change across platform boots or system configuration changes. For example, the PCI device path node only contains a *Device* and a *Function* field. It does not contain a *Bus* field, because the addition of a device with a PCI-to-PCI bridge may modify the bus numbers of other devices in the platform. Instead, the device path for a PCI device is described with one or more PCI device path nodes that describe the path from the PCI root bridge, through zero or more PCI-to-PCI bridges, and finally the target PCI device.

The UEFI Shell is able to display a device path on a console as a string. The conversion of device path nodes to printable strings is not architectural. It is a feature that is implemented in the UEFI Shell that allows developers to view device paths without having to translate manually hex dumps of the device path node data structures. Example 2 shows some example device paths from IA-64 and IA-32 instruction set architecture based platforms. These device paths show standard and extended ACPI device path nodes being used for a PCI root bridge and an ISA floppy controller. PCI device path nodes are used for PCI-to-PCI bridges, PCI video controllers, PCI IDE controllers, and PCI-to-LPC bridges. Finally, IDE messaging device path nodes are used to describe an IDE hard disk, and media device path nodes are used to describe a partition on an IDE hard disk.

```
//
// PCI Root Bridge #0 using an Extended ACPI Device Path
//
Acpi(HWP0002,PNP0A03,0)

//
// PCI Root Bridge #1 using an Extended ACPI Device Path
//
Acpi(HWP0002,PNP0A03,1)

//
// PCI Root Bridge #0 using a standard ACPI Device Path
//
Acpi(PNP0A03,0)

//
// PCI-to-PCI bridge device directly attached to PCI Root Bridge #0
//
Acpi(PNP0A03,0)/Pci(1E|0)

//
// A video adapter installed in a slot on the other side of a PCI-to-PCI
bridge
// that is attached to PCI Root Bridge #0.
//
Acpi(PNP0A03,0)/Pci(1E|0)/Pci(0|0)

//
// A PCI-to-LPC bridge device attached to PCI Root Bridge #0
//
Acpi(PNP0A03,0)/Pci(1F|0)
//
// A 1.44 MB floppy disk controller attached to a PCI-to-LPC bridge
device
// attached to PCI Root Bridge #0
//
Acpi(PNP0A03,0)/Pci(1F|0)/Acpi(PNP0604,0)

//
// A PCI IDE controller attached to PCI Root Bridge #0
//
Acpi(PNP0A03,0)/Pci(1F|1)

//
// An IDE hard disk attached to a PCI IDE controller attached to
// PCI Root Bridge #0
//
Acpi(PNP0A03,0)/Pci(1F|1)/Ata(Secondary,Master)

//
// Partition #1 of an IDE hard disk attached to a PCI IDE controller
attached to
// PCI Root Bridge #0
//
Acpi(PNP0A03,0)/Pci(1F|1)/Ata(Secondary,Master)/HD(Part1,Sig00000000)
```

Example 2. Example Device Paths

2.7.1 How Drivers Use Device Paths

UEFI drivers that manage physical devices must be aware of device paths. When possible, UEFI drivers treat device paths as opaque data structures. Device drivers do not operate on them at all. *Root bridge drivers are required only to produce the device paths for the root bridges, which typically contain only a single ACPI device path node,*

and bus drivers usually just append a single device path node for a child device to the device path of the parent device. The bus drivers should not parse the contents of the parent device path. Instead, a bus driver appends the one device path node that it is required to understand to the device path of the parent device.

For example, a SCSI driver that produces child handles for the disk devices on the SCSI channel will need to build a device path for each disk device. The device path will be constructed by appending a SCSI device path node to the device path of the SCSI controller itself. The SCSI device path node simply contains the Physical Unit Number (PUN) and Logical Unit Number (LUN) of the SCSI disk device, and the driver for a SCSI host adapter is already tracking that information.

The mechanism described above allows the construction of device paths to be a distributed process. The bus drivers at each level of the system hierarchy are required only to understand the device path nodes for their child devices. Bus drivers understand their local view of the device path, and a group of bus drivers from each level of the system bus hierarchy work together to produce complete device paths for the console and boot devices that are used to install and boot operating systems.

There are a number of functions in the **EFI_DEVICE_PATH_UTILITIES** that are available to help manage device paths. Please see the *UEFI Specification* 2.0 section 9.5.2 for full details.

2.7.2 Considerations for Itanium® Architecture

Device paths nodes can be any length, which can actually cause problems on Itanium-based platforms where all data that is accessed must be aligned on proper boundaries. A device path node that is not a multiple of 8 bytes in length will cause the device paths nodes that follow it to be unaligned. Care must be taken when using device paths to make sure an alignment fault is not generated. Device paths that are stored in environment variables are packed on purpose to reduce the amount of nonvolatile storage that is consumed by device paths. These device paths can be unpacked, so each device path node is guaranteed to be on an 8-byte boundary. However, there is no standard way to tell if a device path is packed, unpacked, aligned, or unaligned based on the device path contents alone. Instead, consumers of device paths should always take measures to guarantee that an alignment fault is never generated. The basic technique is to copy a device path node from a potentially unaligned device path into a device path node that is known to be aligned. Then, the device path node contents can be examined or updated and potentially copied back to the original device path. Some example of these operations can be found in section 21.1.

2.7.3 Environment Variables

Device paths are also used when environment variables are built and stored in nonvolatile storage. There are a number of environment variables defined in section 3.2 of the *UEFI 2.0 Specification*. These variables define the following:

- Console input devices
- Console output devices
- Standard error devices
- The drivers that need to be loaded prior to an OS boot

- The boot selections that the platform supports

The UEFI boot manager, UEFI utilities, and UEFI-compliant operating systems manage these environment variables as operating systems are installed and removed from a platform.

2.8 UEFI Driver Model

The *UEFI 2.0 Specification* defines the UEFI Driver Model. Drivers that follow the UEFI Driver Model share the same image characteristics as UEFI applications. However, the model allows UEFI more control over drivers by separating the loading of drivers into memory from the starting and stopping of drivers. Table 7 lists the series of UEFI Driver Model–related protocols that are used to accomplish this separation.

Table 7. Protocols Separating the Loading and Starting/Stopping of Drivers

Protocol	Description
Driver Binding Protocol	Provides functions for starting and stopping the driver, as well as a function for determining if the driver can manage a particular controller. The UEFI Driver Model requires this protocol.
Component Name Protocol	Provides functions for retrieving a human-readable name of a driver and the controllers that a driver is managing. While the *UEFI 2.0 Specification* lists this protocol as optional, the *Developer's Interface Guide for 64-bit Intel Architecture-based Servers* (hereafter referred to as "DIG64 specification" or just "*DIG64*") lists this protocol as required for Itanium-based platforms.
Driver Diagnostics Protocol	Provides functions for executing diagnostic functions on the devices that a driver is managing. While the *UEFI 2.0 Specification* lists this protocol as optional, *DIG64* lists this protocol as required for Itanium-based platforms.
Driver Configuration Protocol	Provides functions that allow the user to configure the devices that a driver is managing. It also allows a device to be placed in its default configuration. While the *UEFI 2.0 Specification* lists this protocol as optional, *DIG64* lists this protocol as required for Itanium-based platforms.

The new protocols are registered on the driver's image handle. In the UEFI Driver Model, the main goal of the driver's entry point is to install theses protocols and exit successfully. At a later point in the system initialization, UEFI can use these protocol functions to operate the driver. A more complex driver may produce more than one instance of the **EFI_DRIVER_BINDING_PROTOCOL**. In this case, the additional instances of the **EFI_DRIVER_BINDING_PROTOCOL** will be installed on new handles. These new handles may also optionally support the Component Name Protocol, Driver Configuration Protocol, and Driver Diagnostics Protocol.

The UEFI Driver Model follows the organization of physical/electrical architecture by defining the following two basic types of UEFI boot time drivers:

- Device drivers
- Bus drivers

A driver that has characteristics of both a device driver and a bus driver is known as a *hybrid driver*.

Device drivers and bus drivers are distinguished by the operations that they perform in the **Start()** and **Stop()** services of the **EFI_DRIVER_BINDING_PROTOCOL**. By walking through the process of connecting a driver to a device, the roles and relationships of the bus drivers and device drivers will become evident; the following sections discuss these two driver types.

2.8.1 Device Driver

The **Start()** service a device driver will install the protocol(s) directly onto the controller handle that was passed into the **Start()** service. The protocol(s) installed by the device driver uses the I/O services that are provided by the bus I/O protocol that is installed on the controller handle. For example, a device driver for a PCI controller would use the services of the PCI I/O Protocol, and a device driver for a USB device would use the service of the USB I/O Protocol. This process is called "consuming the bus I/O abstraction." The following are the main objectives of the device driver:

- Initialize the controller.
- Install an I/O protocol on the device that can be used by the UEFI compliant system firmware to boot the operating system.

It does not make sense to write device drivers for devices that cannot be used to boot a platform. Table 8 lists the standard I/O protocols that the *UEFI 2.0 Specification* defines for different classes of devices.

Table 8. I/O Protocols Used for Different Device Classes

Class of Device	Protocol(s) Used	Defined in this section in *UEFI 2.0 Specification*
Media	**BLOCK_IO_PROTOCOL**	12.7
LAN	**NETWORK_INTERFACE_IDENTIFIER_PROTOCOL** and **UNIVERSAL_NETWORK_DRIVER_INTERFACE**	Appendix E
Console output	**SIMPLE_TEXT_OUTPUT_PROTOCOL**	11.3
Console input	**SIMPLE_INPUT_PROTOCOL**	11.2
Root bus	**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** Note	13.2

Note: *Provided by the system firmware. Drivers that produce this protocol are a special case because it is the first device in the handle database (called a "root" device). Because of this, it does not follow all the normal UEFI Driver Model rules and is not a good example driver to follow when making your own device drivers.*

The fundamental UEFI definition of a device driver is that it does not create any child handles. This capability distinguishes a device driver from a bus driver. This definition has the potential to create some confusion because it is often reasonable to write a driver that creates child handles. This creation will make the driver a bus driver by

definition, but it may not be managing a hardware bus in the classical sense (such as a PCI, SCSI, USB, or Fibre Channel bus).

Just because a device driver does not create child handles does not mean that the device that a device driver is managing will never be a "parent." The protocol(s) produced by a device driver on a controller handle may be consumed by a bus driver that produces child handles. In this case, the controller handle that is managed by a device driver is a parent controller. This scenario happens quite often. For example, the **EFI_USB2_HC_PROTOCOL** is produced by a device driver called the *USB host controller driver.* This protocol is consumed by the bus driver that is called the *USB bus driver* and that creates child handles that contain the **USB_IO_PROTOCOL**. The USB host controller driver that produced the **EFI_USB2_HC_PROTOCOL** has no knowledge of the child handles that are produced by the USB bus driver.

2.8.2 Bus Driver

A bus driver is nearly identical to a device driver except that it creates child handles. This ability leads to several added features and responsibilities for a bus driver that will be addressed in detail throughout this document. For example, device drivers do not need to concern themselves with searching the bus.

Just like a device driver, the **Start()** function of a bus driver will consume the parent bus I/O abstraction(s) and produce new I/O abstractions in the form of protocols. For example, the PCI *bus driver* consumes the services of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** and uses these services to scan a PCI bus for PCI controllers. Each time that a PCI controller is found, a child handle is created and the **EFI_PCI_IO_PROTOCOL** is installed on the child handle. The services of the **EFI_PCI_IO_PROTOCOL** are implemented using the services of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. As a second example, the USB bus driver uses the services of the **EFI_USB2_HC_PROTOCOL** to discover and create child handles that support the **EFI_USB_IO_PROTOCOL** for each USB device on the USB bus. The services of the **EFI_USB_IO_PROTOCOL** are implemented using the services of the **EFI_USB2_HC_PROTOCOL**.

The following are the main objectives of the bus driver:

- Initialize the bus controller.
- Determine how many children to create. For example, the PCI bus driver may discover and enumerate all PCI devices on the bus or only a single PCI device that is being used to boot.

  How a bus driver handles this step creates a basic subdivision in the types of bus drivers. A bus driver can do one of the following:
  - Create handles for all child controllers on the first call to **Start()**.
  - Allow the handles for the child controllers to be created across multiple calls to **Start()**.

  This second type of bus driver is very useful because it reduces the platform boot time. It allows a few child handles or even a single child handle to be created. On buses that take a long time to enumerate their children (for example, SCSI and Fibre Channel), multiple calls to **Start()** can save a large amount of time when booting a platform.

- Allocate resources and create a child handle in the UEFI handle database for one or more child controllers.
- Install an I/O protocol on the child handle that abstracts the I/O operations that the controller supports (such as the PCI I/O Protocol or the USB I/O Protocol).
- If the child handle represents a physical device, then install a Device Path Protocol (see chapter 9 in the *UEFI 2.0 Specification*).
- Load drivers from option ROMs if present. The PCI bus driver is the only bus driver that loads from option ROMs so far.

Some common examples of UEFI bus drivers include the following:

- PCI bus driver
- USB bus driver
- SCSI bus drivers

Because bus drivers are defined as drivers that produce child handles, there are some other drivers, *such as the following,* that unexpectedly qualify as bus drivers:

Serial driver:

Creates a child handle and extends the Device Path Protocol to include a **`Uart()`** messaging device path node.

LAN driver:

Creates a child handle and extends the Device Path Protocol to include a **`Mac()`** address messaging device path node.

Graphics driver:

Creates a child handle for each physical video output and any logical video output that is a combination of 2 or more physical video outputs. Graphics drivers do not extend the Device Path Protocol.

2.9 Driver Connection Process

All UEFI drivers that adhere to the UEFI Driver Model follow the same basic procedure. When the driver is loaded, it will install a Driver Binding Protocol on the image handle from which it was loaded. It may also update a pointer to the **`EFI_LOADED_IMAGE`** Protocol's **`Unload()`** service and install Component Name Protocol so its name is visible to any operator. The driver will then exit from the entry point, leaving the code resident in system memory.

The Driver Binding Protocol provides a version number and the following three services:

- **`Supported()`**
- **`Start()`**
- **`Stop()`**

These services are available on the driver's image handle after the entry point is exited. Later on when the system is "connecting" drivers to devices in the system, the driver's

Driver Binding Protocol **Supported()** service is called. The **Supported()** service is passed a controller handle. Quickly, the **Supported()** function will examine the controller handle to see if it represents a device that the driver knows how to manage. If so, it will return **EFI_SUCCESS**. The system will then start the driver by calling the driver's **Start()** service, passing in the supported controller handle. The driver can later be disconnected from a controller handle by calling the **Stop()** service.

This section will walk the reader through the connection process that a platform may undergo to connect the devices in the system with the drivers that are available in the system. This process will appear complex at first, but as the process continues, it will become evident that the same simple procedures are used to accomplish the complex task. This description does not go into all the details of the connection process but explains enough that the role of various drivers in the connection process can be clearly understood. This knowledge is fundamental to designing new UEFI drivers.

The EFI Boot Service **ConnectController()** clearly demonstrates the power of the UEFI Driver Model. The UEFI Shell command **connect** directly exposes much of the functionality of this boot service and provides a convenient way to explore the flexibility and control offered by **ConnectController()**.

2.9.1 ConnectController()

By passing the handle of a specific controller into **ConnectController()**, UEFI will follow a specific process to determine which driver(s) will manage the controller.

For reference, the following is the definition of **ConnectController()**:

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

The connection is a two-phase process, as follows:

First phase: Construct an ordered list of driver handles.

Second phase: Try to connect the drivers to a controller in priority order from first to last.

Table 9 lists the steps for phase one, which are known as the *driver connection precedence rules.* Much of this information is in section 6.3.1 of the *UEFI 2.0 Specification* where the EFI Boot Service **ConnectController()**is discussed.

Table 9. Connecting Controllers: Driver Connection Precedence Rules

Step	Type of Override	Description
1	Context override	The parameter *DriverImageHandle* is an ordered list of image handles. The highest-priority driver handle is the first element of the list, and the lowest-priority driver handle is the last element of the list. The list is terminated with a **NULL** driver handle. This parameter is usually **NULL** and is typically used only to debug new drivers from the UEFI Shell. These drivers are placed at the head of the ordered list of driver handles.
2	Platform driver override	If an **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL** instance is present in the system, then the **GetDriver()** service of this protocol is used to retrieve an ordered list of image handles for *ControllerHandle*. The first driver handle returned from **GetDriver()** has the highest precedence, and the last driver handle returned from **GetDriver()** has the lowest precedence. The ordered list is terminated when **GetDriver()** returns **EFI_NOT_FOUND**. It is legal for no driver handles to be returned by **GetDriver()**. Any driver handles obtained from the **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL** are appended to the end of the ordered list of driver handles from the context override (step #1 above).
3	Bus specific driver override	If there is an instance of the **EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL** attached to *ControllerHandle*, then the **GetDriver()** service of this protocol is used to retrieve an ordered list of driver handles for *ControllerHandle*. The first driver handle returned from **GetDriver()** has the highest precedence, and the last driver handle returned from **GetDriver()** has the lowest precedence. The ordered list is terminated when **GetDriver()** returns **EFI_NOT_FOUND**. It is legal for no driver handles to be returned from **GetDriver()**. Any driver handles obtained from the **EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL** are appended to the end of the ordered list of driver handles that was produced from the context override and platform driver override (steps 1 and 2). In practice, this precedent option allows the UEFI drivers that are stored in an option ROM of a PCI adapter to manage the PCI adapter. Even if drivers with higher versions are available, this rule exists to make sure that if a particular driver on a card requires a hardware modification to go with it, the driver will not get replaced by later versions of the driver that may not work with the hardware modification.
4	Driver binding search	The list of available driver handles can be found by using the boot service **LocateHandle()** with a *SearchType* of *ByProtocol* for the GUID of the **EFI_DRIVER_BINDING_PROTOCOL**. From this list, the driver handles found in steps 1, 2, and 3 above are removed. The remaining driver handles are sorted from highest to lowest based on the *Version* field of the **EFI_DRIVER_BINDING_PROTOCOL** instance that is associated with each driver handle. Any driver handles that are obtained from this search are appended to the end of the ordered list of driver handles started in steps 1, 2, and 3. In practice, this sorting means that a PCI adapter that does not have an UEFI driver in its option ROM will be managed by the driver with the highest *Version* number.

Phase two of the connection process is to check each driver in the ordered list to see if it supports the controller. This check is done by calling the **Supported()** service of the driver's Driver Binding Protocol and passing in the *ControllerHandle* and the *RemainingDevicePath*. If successful, the **Start()** service of the driver's Driver Binding Protocol is called and passes in the *ControllerHandle* and *RemainingDevicePath*.. Each driver in the list is given an opportunity to connect, even if a prior driver connected successfully. However, if a driver with higher priority had already connected and opened the parent I/O protocol with exclusive access, the other drivers would not be able to connect.

One reason this type of connection process is used is because the order in which drivers are installed into the handle database is not deterministic. Drivers can be unloaded and reloaded later, which changes the order of the drivers in the handle database.

These precedent rules assume that the relevant drivers to be considered are loaded into memory. This case may not be true for all systems. Large systems, for example, may limit "bootable" devices to a subset of the total number of devices in the system.

The **ConnectController()** function can be called several times during the initialization of EFI. It is called before launching the boot manager to set the consoles.

2.9.2 Loading UEFI Option ROM Drivers

The following is an interesting use case that tests these precedence rules. Assume that the following three identical adapters are in the system:

- Adapter A: UEFI driver Version 0x10
- Adapter B: UEFI driver Version 0x11
- Adapter C: No UEFI driver

These three adapters have UEFI drivers in the option ROM as defined below. When UEFI connects, the drivers control the devices as follows:

- UEFI driver Version 0x10 manages Adapter A.
- UEFI driver Version 0x11 manages Adapter B and Adapter C.

If the user then goes into the UEFI Shell and soft loads UEFI driver version 0x12, nothing will change until the existing drivers are disconnected and a reconnect is performed. This reconnection can be done in a variety of ways, but the UEFI Shell command **reconnect -r** is the easiest. Now the drivers control the devices as follows:

- UEFI driver Version 0x10 manages Adapter A.
- UEFI driver Version 0x11 manages Adapter B.
- UEFI driver Version 0x12 manages Adapter C.

An OEM can override this logic by implementing the Platform Driver Override Protocol.

2.9.3 DisconnectController()

DisconnectController() performs the opposite of **ConnectController()**. It requests drivers that are managing a controller to release the controller.

2.10 Platform Initialization

Figure 7 shows the sequence of events that occur when an EFI-based system is booted. The following sections will describe each of these events in detail and how they relate to UEFI drivers.

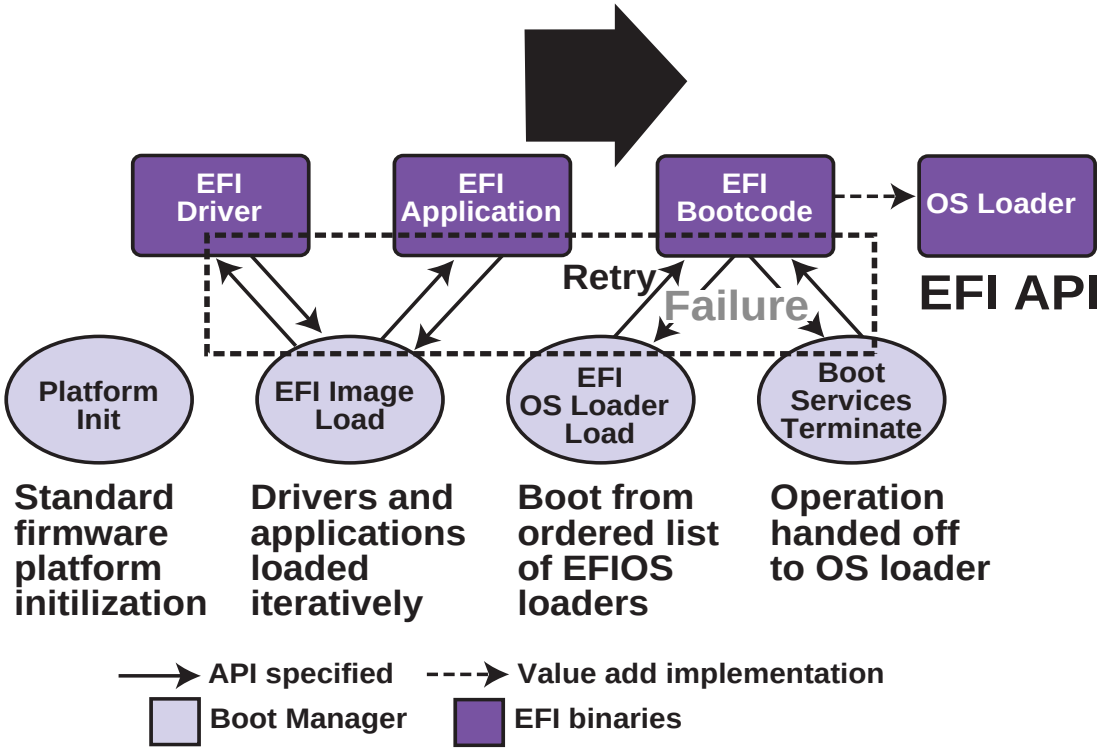

Figure 7 Booting Sequence

Figure 8 shows a possible system configuration. Each box represents a physical device (called a *controller*) in the system. Before the first UEFI connection process is performed, none of these devices is registered in the handle database. The following sections describe the steps that *UEFI compliant firmware* follows to initialize a platform and how drivers are executed, handles are created, and protocols are installed.

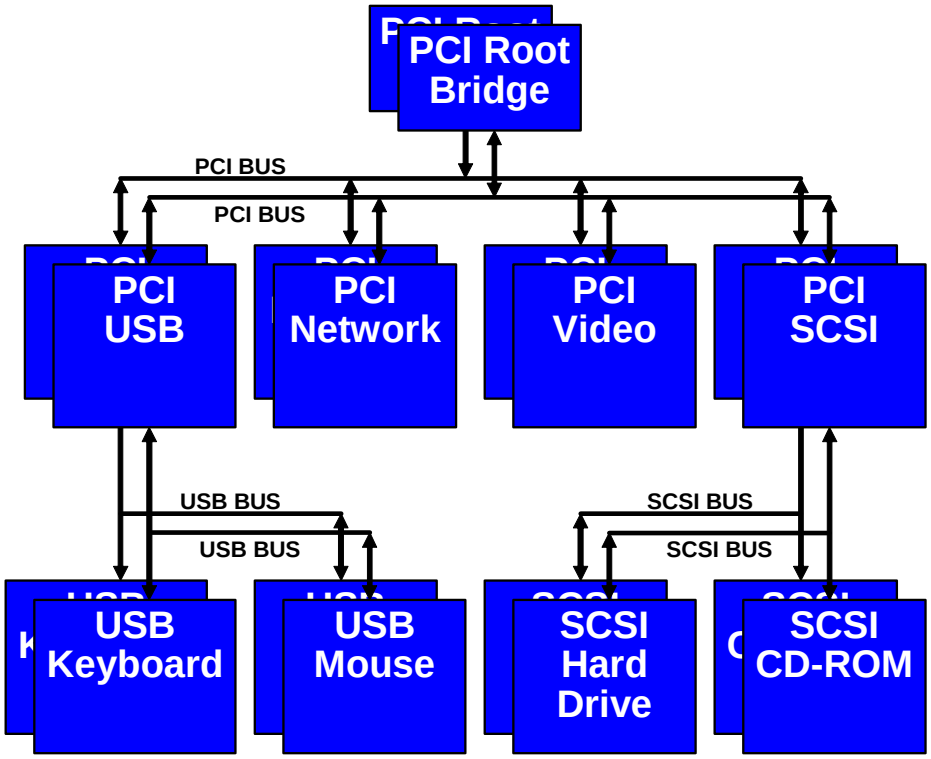

Figure 8. Sample System Configuration

At some point early in the boot process, UEFI initialization will create handles and install the EBC Protocol and the Decompression Protocol in the handle database. These service protocols will be needed to run UEFI drivers that may be compressed or compiled in an EBC format. For example, the handle database as viewed with the **dh** UEFI Shell command might look like the following:

Handle Database

```
…
  6: Ebc
…
  9: Image(Decompress)
  A: Decompress
```

2.10.1 Connecting PCI Root Bridge(s)

During UEFI compliant firmware initialization, the system will load a root bridge driver for the root device, typically the PCI root bridge driver, with **LoadImage()**. Like all drivers, as it is loaded, UEFI firmware will create a handle in the handle database and attach an instance of the **EFI_LOADED_IMAGE_PROTOCOL** with the unique image

information for the PCI root bridge driver. Because this driver is the system root driver, it does not follow the UEFI Driver Model. When it is loaded, it does not install the following:

- Driver Binding Protocol
- Component Name Protocol
- Driver Diagnostics Protocol
- Driver Configuration Protocol

Instead, it immediately uses its knowledge about the platform architecture to create handles for each PCI root bridge. This example shows only one PCI root bridge, but there can be many PCI root bridges in a system, especially in a data center server. It installs the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** and an **EFI_DEVICE_PATH_PROTOCOL** on each handle. By not installing the Driver Binding Protocol, the PCI root bridge prevents itself from being disconnected or reconnected later on. For example, the handle database as viewed with the **dh** UEFI Shell command might look like the following after the PCI root bridge driver is loaded and executed. This example shows an image handle that is a single controller handle with a PCI Root Bridge I/O Protocol and the Device Path Protocol.

Handle Database

```
…
  B: Image(PcatPciRootBridge)
  C: DevIo PciRootBridgeIo DevPath (Acpi(HWP0002,0,PNP0A03))
```

Note: *PNP0A03 may appear in either _HID or _CID of the PCI root bridge device path node. This example is one where it is not in _HID.*

OS loaders need to access the boot devices to boot the OS. Such devices must have a Device Path Protocol, because the OS loader uses this protocol to determine the location of the boot device. Here at the root device, the Device Path Protocol contains a single ACPI node **Acpi(HID, UID)** or **Acpi(HID, UID, CID)**. This node points the OS to the place in the ACPI name space where the PCI root bridge is completely described. The **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** provides PCI functions that are used by the PCI bus driver that is described in next section.

2.10.2 Connecting the PCI Bus

EFI initialization will continue by loading the PCI bus driver. As its entry point is executed, the PCI bus driver installs the Driver Binding Protocol and Component Name Protocol. For example, the handle database as viewed with the **dh** UEFI Shell command might look like the following after the PCI bus driver is loaded and executed. It contains one new driver image handle with the Loaded Image Protocol, Driver Binding Protocol, and Component Name Protocol. Because this driver does follow the UEFI Driver Model, no new controller handles are produced when the driver is loaded and executed. They will not be produced until the driver is connected.

Handle Database

```
…
14: Image(PciBus) DriverBinding ComponentName
```

At some point later in the initialization process, UEFI compliant firmware will use **ConnectController()** to attempt to connect the PCI root bridge controller(s) (handle #C). The system has several priority rules for determining what driver to try first, but in this case it will search the handle database for driver handles (handles with the Driver Binding Protocol). The search will find handle #14 and call the driver's **Supported()** service, passing in *controller handle #C.* The PCI bus driver requires the Device Path Protocol and PCI Root Bridge I/O Protocol to be started, so the **Supported()** service will return **EFI_SUCESS** when those two protocols are found on a handle. After receiving **EFI_SUCCESS** from the **Supported()** service, **ConnectController()** will then call the **Start()** service with the same controller handle (#C).

Because the PCI bus driver is a bus driver, the **Start()** service will use the PCI Root Bridge I/O Protocol functions to search the PCI bus for all PCI devices. For each PCI device/function that the PCI bus driver finds, it will create a child handle and install an instance of the PCI I/O Protocol on the handle. The handle is registered in the handle database as a "child" of the PCI root bridge controller. The PCI driver will also copy the device path from the parent PCI root bridge device handle and append a new PCI device path node **Pci(Dev|Func)**. In cases where the PCI driver discovers a PCI-to-PCI bridge, the devices below the bridge will also be added as children to the bridge. In these cases, an extra PCI device path node will be added for each PCI-to-PCI bridge in between the PCI root bridge and the PCI device. For example, the handle database as viewed with the **dh** UEFI Shell command might look like the following after the PCI bus driver is connected to the PCI root bridge. It shows the following:

- Nine PCI devices were discovered.
- The PCI device on handle #1B has an option ROM with an UEFI driver.
- That UEFI driver was loaded and executed and is shown as handle #1C.

Also notice that a single PCI card may have several UEFI handles if they have multiple PCI functions.

Handle Database

```
…
 16: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(1|0))
 17: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(1|1))
 18: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(2|0))
 19: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(2|1))
 1A: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(2|2))
 1B: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(3|0))
 1C: Image(Acpi(HWP0002,0,PNP0A03)/Pci(3|0)) DriverBinding
 1D: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(4|0))
 1E: PciIo DevPath (Acpi(HWP0002,100,PNP0A03)/Pci(1|0))
 1F: PciIo DevPath (Acpi(HWP0002,100,PNP0A03)/Pci(1|1))
```

2.10.3 Connecting Consoles

At this point during the UEFI compliant firmware initialization, the firmware does not have a "console" device configured. This absence is because it is often a PCI device, and we have been waiting for the PCI bus driver to provide device handles for the console(s). Most UEFI compliant platforms will now follow a console connection strategy to connect the consoles in a consistent manner that is defined by the platform. This way the platform will be able to display messages to all of the selected consoles through the standard UEFI mechanisms. Prior to this point, the platform messages, if any, were being displayed by platform-specific methods.

2.10.4 Console Drivers

UEFI consoles include the following:

- Graphics devices
- Keyboards
- Mice
- Serial ports

Some systems may have other devices.

The following table shows an example of drivers that are installed in the handle database by the platform firmware. These drivers are in the system ROM because they are core devices built into the motherboard. If the devices were PCI cards, the USB host controller driver and Graphics Output may have been found in the UEFI compliant Option ROMs on each of those cards.

Table 10. UEFI Console Drivers

Class of Driver	Type of Driver	Driver Name	Description and Example
USB Console	USB host controller driver	UsbOhci or UsbUhci	Consumes the PCI I/O Protocol and produces the USB Host Controller Protocol. **25: Image(UsbOhci) DriverBinding ComponentName**
	USB bus driver	UsbBus	Consumes the USB Host Controller Protocol and produces the USB I/O Protocol. **26: Image(UsbBus) DriverBinding ComponentName**
	USB keyboard driver	UsbKb	Consumes the USB I/O Protocol and produces the Simple Input Protocol. **27: Image(UsbKb) DriverBinding ComponentName**
	USB mouse driver	UsbMouse	Consumes the USB I/O Protocol and produces the Simple Pointer Protocol. **28: Image(UsbMouse) DriverBinding ComponentName**
Graphics	Graphics Output	GOP	Consumes the PCI I/O Protocol and produces the Graphics Output Protocol. **2E: Image(GOP) DriverBinding ComponentName**
	Graphics console driver	GraphicsConsole	Consumes the Graphics Output Protocol and produces the Simple Text Output Protocol. **2D: Image(GraphicsConsole) DriverBinding ComponentName**

Serial	Serial 16550 UART driver	Serial16550	Consumes the ISA I/O Protocol and produces the Serial I/O Protocol. **30: Image(Serial16550) DriverBinding ComponentName**
	Serial terminal driver	Terminal	Consumes the Serial I/O Protocol and produces the Simple Input and Simple Text Output Protocols. **31: Image(Terminal) DriverBinding ComponentName**
Generic Console	Platform console management driver	ConPlatform	This driver is a bit unique in that a single set of driver code adds two driver handles, one for the "Console Out" and another for the "Console In." **32: Image(ConPlatform) DriverBinding ComponentName** **33: DriverBinding ComponentName**

	Console splitter driver	ConSplitter	This driver may not be present on all platforms. It will combine the various selected input and output devices for the following four basic UEFI user devices: ConIn ConOut ErrOut PointerIn It also installs multiple driver handles for a single set of driver code. It installs driver handles to manage ConIn, ConOut, ErrOut, and PointerIn devices. The entry point of this driver creates virtual handles for ConIn, ConOut, and StdErr, respectively, that are called the following: PrimaryConIn PrimaryConOut PrimaryStdErr The virtual handles will always exist even if no console exists in the system. These virtual handles are needed to support hot-plug devices such as USB. **34: Image(ConSplitter) DriverBinding ComponentName** **35: DriverBinding ComponentName** **36: DriverBinding ComponentName** **37: DriverBinding ComponentName** **38: Txtout PrimaryStdErr** **39: Txtin SimplePointer PrimaryConIn** **3A: Txtout PrimaryConOut GraphicsOutput**

2.10.5 Console Variables

After loading these drivers in the handle database, the platform can connect the console devices that the user has selected. The device paths for these consoles will be stored in the *ConIn*, *ConOut*, and *ErrOut* global UEFI variables (see section 3.2 in the *UEFI 2.0 Specification*). For the purpose of this example, the variables have the following device paths:

```
ErrOut = Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600N81)/
VenMsg(Vt100+);Acpi(HWP0002,0,PNP0A03)/Pci(4|0)

ConOut = Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600N81)/
VenMsg(Vt100+);Acpi(HWP0002,0,PNP0A03)/Pci(4|0)

ConIn  = Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600N81)/
VenMsg(Vt100+)
```

Note: *Note: The* `ErrOut` *and* `ConOut` *variables are compound device paths indicating that the UEFI output is mirrored on two devices. This work is done by the console splitter driver when it is connected. The two devices are a serial terminal and a PCI video controller.*

Note: *Note: The* `ConIn` *variable contains a device path to a serial terminal.*

Note: *Note: The* `ErrOut` *variable is typically the same as the* `ConOut` *variable, but could be redirected to another device. It is important to check when developing UEFI drivers because the* ***`DEBUG()`*** *output is typically directed to the* `ErrOut` *device and it may not always be the same device as the* `ConOut` *device. In this case, the two devices are a serial terminal and a PCI video controller.*

2.10.6 ConIn

Now the system will connect the console devices using the device paths in the *`ConIn`*, *`ConOut`*, and *`ErrOut`* global UEFI variables. The *`ConIn`* connection process will be discussed first.

```
ConIn  = Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600 N81)/
VenMsg(Vt100+)
```

The UEFI connection process will search for the device in the handle database that has a device path the matches the following the closest.

```
Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600 N81)/VenMsg(Vt100+)
```

It will find handle #17 as the closest match. The portion of the device path that did not match (**`Uart(9600 N81)/VenMsg(Vt100+)`**) is called the *remaining device path.*

```
17: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(1|1))
```

EFI will call **`ConnectController()`**, passing in handle #17 and the remaining device path. The connection code will construct a list of all the drivers in the system and call each driver, passing handle #17 and the remaining device path into the **`Supported()`** service. The only driver installed in the handle database that will return **`EFI_SUCCESS`** for this device handle is handle #30:

```
30: Image(Serial16550) DriverBinding ComponentName
```

Now that **`ConnectController()`** has found a driver that supports handle #17, it will pass device handle #17 and the remaining device path **`Uart(9600 N81)/ VenMsg(Vt100+)`** into the serial driver's **`Start()`** service. The serial driver will open the PCI I/O Protocol on handle #17 and create a new child handle. The following will be installed onto the new child handle:

- **`EFI_SERIAL_IO_PROTOCOL`** (defined in section 11.6 of the *UEFI 2.0 Specification*)

- **EFI_DEVICE_PATH_PROTOCOL**

The device path for the child handle is generated by making a copy of the device path from the parent and appending the serial device path node **Uart(9600 N81)**. Handle #3B shown below is the new child handle.

```
3B: SerialIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/
Uart(9600 N81))
```

That first call to **ConnectController()** has now been completed, but the Device Path Protocol on handle #3B does not completely match the *ConIn* device path, so the connection process will be repeated. This time the closest match for **Acpi(HWP0002,0)/Pci(1|1)/Uart(9600 N81)/VenMsg(Vt100+)** is the newly created device handle #3B. Now the remaining device path is **VenMsg(Vt100+)**. The search for a driver that supports handle #3B will find the terminal driver, returning **EFI_SUCCESS** from the **Supported()** service.

```
31: Image(Terminal) DriverBinding ComponentName
```

This driver's **Start()** service will open the **EFI_SERIAL IO_PROTOCOL**, create a new child handle, and install the following:

- **EFI_SIMPLE_TEXT_INPUT_PROTOCOL**
- **EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL**
- **EFI_DEVICE_PATH_PROTOCOL**

The console protocols are defined in chapter 11 of the *UEFI 2.0 Specification*. The device path will be generated by making a copy of the device path from the parent and appending the terminal device path node **VenMsg(Vt100+)**. VT100+ was chosen because that terminal type was specified in the remaining device path that was passed into the **Start()** service. Handle #3C show below is the new child handle.

```
3C: Txtin Txtout DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/
Uart(9600 N81)/VenMsg(Vt100+))
```

The process still has not completely matched the *ConIn* device path, so the connection process will be repeated again. This time there is an exact match for **Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600 N81)/VenMsg(Vt100+)** with the newly created child handle #3C. Searching for a driver that will support this controller will find two driver handles that return **EFI_SUCCESS** to the **Supported()** service. The two driver handles are from the platform console management driver:

```
32: Image(ConPlatform) DriverBinding ComponentName
33: DriverBinding ComponentName
```

Driver #32 will install a *ConOut* device GUID on the handle if the device path is listed in the *ConOut* global UEFI variable. In our example, this case is true. Driver #32 will also install an *ErrOut* device GUID on the handle if the device path is listed in the *ErrOut* global UEFI variable. This case is also true in our example. Therefore, handle #3C will have two new protocols on it: *ConOut* and *StdErr*.

```
3C: Txtin Txtout ConOut StdErr DevPath (..art(9600 N81)/
VenMsg(Vt100+))
```

Driver #33 will install a *ConIn* device GUID on the handle if the device path is listed in the *ConIn* global UEFI variable (which it will because we started this connection process that way), so handle #3C will have the *ConIn* protocol attached.

```
3C: Txtin Txtout ConIn ConOut StdErr DevPath (..art(9600
N81)/VenMsg(Vt100+))
```

EFI uses these three protocols (*ConIn*, *ConOut*, and *ErrOut*) to mark devices in the system, which have been selected by the user as *ConIn*, *ConOut*, and *StdErr*. These protocols are actually just a GUID without any services or data.

There are three other driver handles that will return **EFI_SUCCESS** from the **Supported()** service. These driver handles are from the console splitter drivers for the *ConIn*, *ConOut*, and *ErrOut* devices in the system. There is a fourth console splitter driver handle (which is not used on this handle) for devices that support the Simple Pointer Protocol. The three driver handles are listed below:

```
34: Image(ConSplitter) DriverBinding ComponentName
36: DriverBinding ComponentName
37: DriverBinding ComponentName
```

Remember that when the console splitter driver was first loaded, it created three virtual handles. It marked these three handles with protocols *PrimaryStdErr*, *PrimaryConIn*, and *PrimaryConOut*. These protocols are only GUIDs. They do not contain any services or data.

```
38: Txtout PrimaryStdErr
39: Txtin SimplePointer PrimaryConIn
3A: Txtout PrimaryConOut GraphicsOutput
```

The console splitter driver's **Supported()** service for handle #34 examines the handle #3C for a *ConIn* Protocol. Having found it, it returns **EFI_SUCCESS**. The **Start()** service will then open the *ConIn* protocol on handle #3C such that the *PrimaryConIn* device handle #39 becomes a child controller and starts aggregating the **SIMPLE_INPUT_PROTOCOL** services.

The same thing happens for handle #36 with *ConIn*, except that the **SIMPLE_TEXT_OUTPUT_PROTOCOL** functionality on handle #3C is aggregated into the **SIMPLE_TEXT_OUTPUT_PROTOCOL** on the *PrimaryConOut* handle #3A.

Handle #37 with *StdErr* also does the same thing; the **SIMPLE_TEXT_OUTPUT_PROTOCOL** functionality on handle #3C is aggregated into the **SIMPLE_TEXT_OUTPUT_PROTOCOL** on the *PrimaryStdErr* handle #38.

The connection process has now been completed for *ConIn* because the device path that completely matched the *ConIn* device path and all the console-related services has been installed.

2.10.7 ConOut

As with *ConIn*, UEFI compliant firmware will connect the *ConOut* devices using the device paths in the *ConOut* global UEFI variable. If *ConIn* was not complicated enough, the *ConOut* global UEF device path in this example is a compound device path and indicates that the *ConOut* device is being mirrored with the console splitter driver to two separate devices.

```
ConOut = Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600 N81)/
VenMsg(Vt100+);Acpi(HWP0002,0,PNP0A03)/Pci(4|0)
```

The UEFI connection process will search the handle database for a device path that matches the first device path in the *ConOut* variable:
Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600 N81)/VenMsg(Vt100+)

Luckily, the device path already exists on handle #3C in its entirety thanks to the connection work done for *ConIn*.
3C: Txtin Txtout ConIn ConOut StdErr DevPath (..art(9600 N81)/VenMsg(Vt100+))

EFI will perform a **ConnectController()** on handle #3C. Because this step was done previously with *ConIn*, there is nothing more to be done here.

The connection process has not yet been completed for *ConOut* because the device path is a compound device path and a second device needs to be connected:
Acpi(HWP0002,0,PNP0A03)/Pci(4|0)

The UEFI connection process will search the handle database for a device path that matches **Acpi(HWP0002,0,PNP0A03)/Pci(4|0)**. The device path already exists in its entirety on handle #1C and was created by the PCI bus driver when it started and exposed the PCI devices.
1C: PciIo DevPath (Acpi(HWP0002,0,PNP0A03)/Pci(4|0))

EFI will perform a **ConnectController()** on handle #1C. Note that the device path is a complete match, so there is no remaining device path to pass in this time. **ConnectController()** constructs the prioritized list of drivers in the system and calls the **Supported()** service for each one, passing in the device handle #1C. The only driver that will return **EFI_SUCCESS** is the GraphicsOutput driver.
2E: Image(CirrusLogic5430) DriverBinding ComponentName

ConnectController() will **Start()** this driver and it will consume the device's **EFI_PCI_IO_PROTOCOL** and install the **EFI_GRAPHICS_OUTPUT_PROTOCOL** onto the *device handle* #1C.
1C: PciIo GraphicsOutput DevPath (Acpi(HWP0002,0,PNP0A03)/ Pci(4|0))

ConnectController() will continue to process its list of drivers and will find that the "GraphicsConsole" driver's **Supported()** service will return **EFI_SUCCESS**.
2D: Image(GraphicsConsole) DriverBinding ComponentName

Next the graphics console driver's **Start()** service will consume the **EFI_GRAPHICS_OUTPUT_PROTOCOL** and produce the **EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL** on the same *device handle* #1C.
1C: Txtout PciIo GraphicsOutput DevPath (Acpi(HWP0002,0,PNP0A03)/ Pci(4|0))

ConnectController() will continue to process its list of drivers. This time, searching for a driver that will support this controller will find two driver handles that return **EFI_SUCCESS** to the **Supported()** service. These two driver handles are from the platform console management driver:
32: Image(ConPlatform) DriverBinding ComponentName

Driver handle #32 will install a *ConOut* device GUID on the handle if the device path is listed in the *ConOut* global UEFI variable. In this example, this case is true. Driver #32 will also install an *ErrOut* device GUID on the handle if the device path is listed in the *ErrOut* global UEFI variable. This case is also true in the example. Therefore, handle #1C has two new protocols on it: *ConOut* and *ErrOut*.

```
1C: Txtout PciIo ConOut StdErr DevPath
Acpi(HWP0002,0,PNP0A03)/Pci(4|0))
```

These two protocols (*ConOut* and *ErrOut*) are used to mark devices in the system that has been selected by the user as *ConOut* and *StdErr*. These protocols are actually just a GUID without any functions or data.

There are two other driver handles that will return **EFI_SUCCESS** to the **Supported()** service. These driver handles are from the console splitter driver for the *ConOut* and *ErrOut* devices in the system.

```
36: DriverBinding ComponentName
37: DriverBinding ComponentName
```

Remember that when the console splitter driver was first loaded, it created three virtual handles. It marked these three handles with protocols *PrimaryStdErr*, *PrimaryConIn*, and *PrimaryConOut*. These protocols are only GUIDs. They do not contain any services or data.

```
38: Txtout PrimaryStdErr
39: Txtin SimplePointer PrimaryConIn
3A: Txtout PrimaryConOut GraphicsOutput
```

The console splitter driver's **Supported()** service for *driver handle* #36 examines the handle #1C for a *ConOut* Protocol. Having found it, it returns **EFI_SUCCESS**. The **Start()** service will then open the *ConOut* protocol on *device handle* #1C such that the *PrimaryConOut device handle* #3A becomes a child controller and starts aggregating the **SIMPLE_TEXT_OUTPUT_PROTOCOL** services.

The same thing happens for *driver handle* #37 with *StdErr*; the **SIMPLE_TEXT_OUTPUT_PROTOCOL** functionality on *device handle* #1C is aggregated into the **SIMPLE_TEXT_OUTPUT_PROTOCOL** on the *PrimaryStdErr device handle* #38.

2.10.8 ErrOut

In this example, *ErrOut* is the same as *ConOut*. So the connection process for *ConOut* is executed one more time.

```
ErrOut = Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600 N81)/
VenMsg(Vt100+);Acpi(HWP0002,0,PNP0A03)/Pci(4|0)
```

2.10.9 Loading Other Core Drivers

After connecting the consoles, most platforms will typically load some additional core drivers, but this step is dependent on platform policy. For example, the *EFI Developers Kit* contains the following drivers:

- Disk media support drivers (DiskIo, Partition, Fat)

- Platform media drivers (for example, platform-specific bus drivers for SCSI and IDE; for this example, it will load "XyzScsi")
- Core LAN drivers (XyzUndi)
- LAN support drivers (BiosSnp16, Snp3264, PxeBc, PxeDhcp4, BIS, MNP)
- FPSWA (Itanium architecture only)

Remember that these drivers are only loaded. They are not connected to controllers until **ConnectController()** is called.

2.10.10 Boot Manager Connect ALL

On some platforms, the UEFI boot manager may connect all drivers to all devices at this point in the platform initialization sequence. However, platform firmware can choose to connect the minimum number of drivers and devices that is required to establish consoles and gain access to the boot device. Performing the minimum amount of work is recommended to enable shorter boot times.

If the platform firmware chooses to go into a "platform configuration" mode, then all the drivers should be connected to all devices. The following algorithm is used to connect all drivers to all devices.

A search is made of the handle database for all root controller handles. These handles do not have a Driver Binding Protocol or the Loaded Image Protocol. They will have a Device Path Protocol, and they will not have any parent controllers. **ConnectController()** is called with the *Recursive* flag set to **TRUE** and a *RemainingDevicePath* of **NULL** for each of the root controllers. These settings cause all children to be produced by all bus drivers. As each bus is expanded, if the bus supports storage devices for UEFI drivers, then additional UEFI drivers will be loaded from those storage devices (for example, option ROMs on PCI adapters). This process is recursive. Each time a child handle is created, **ConnectController()** is called again on that child handle, so all of those handle's children will be produced. When the process is complete, then the entire tree of boot devices in the system hierarchy will be present in the handle database.

2.10.11 Boot Manager Driver Option List

The UEFI boot manager loads the drivers that are specified by the *DriverOrder* and *Driver####* environment variables. These environment variables are discussed in sections 3.1 and 3.2 of the *UEFI 2.0 Specification*. The UEFI Shell command **bcfg** can be used to add, remove, and display drivers to this list.

Before the UEFI boot manager loads each driver, it will use the device path stored in the *Driver####* variable to connect the controllers and drivers that are required to access the driver option. This process is exactly the same as the process used for the console variables *ErrOut*, *ConOut*, and *ConIn*.

If any driver in the *DriverOrder* list has a load attribute of **LOAD_OPTION_FORCE_RECONNECT**, then the UEFI boot manager will use the **DisconnectController()** and **ConnectController()** boot services to disconnect and reconnect all the drivers in the platform. This load attribute allows the newly loaded drivers to be considered in the driver connection process.

For example, if no driver in the *DriverOrder* list has the **LOAD_OPTION_FORCE_RECONNECT** load attribute, then it would be possible for a built-in system driver with a low version number to manage a device. Then, after loading a newer driver with a higher version number from the *DriverOrder* list, the driver with the lower version number will still manage the same device. However, if the newer driver in the *DriverOrder* list has a load attribute of **LOAD_OPTION_FORCE_RECONNECT**, then the UEFI boot manager will disconnect and reconnect all the controllers, so the driver with the highest version number will manage the same device that the lower versioned driver was managing. Drivers that are added to the *DriverOrder* list should not set the **LOAD_OPTION_FORCE_RECONNECT** attribute unless they have to because the disconnect and reconnect process will increase the boot time.

2.10.12 Boot Manager BootNext

After connecting any drivers in the *DriverOrder* list, the UEFI boot manager will attempt to boot the boot option that is specified by the *BootNext* environment variable. This environment variable is discussed in sections 3.1 and 3.2 of the *UEFI 2.0 Specification*. This variable typically is not set, but if it is, the firmware will delete the variable and then attempt to load the boot option that is described in the *Boot####* variable pointed to by *BootNext*.

Before the UEFI boot manager boots the boot option, it will use the device path stored in the *Boot####* variable to connect the controllers and drivers that are required to access the boot option. This process is exactly the same as the process that is used for the console variables *ErrOut*, *ConOut*, and *ConIn*.

2.10.13 Boot Manager Boot Option List

The UEFI boot manager will display the boot option menu and if the auto-boot *TimeOut* environment variable has been set, then the first boot option will be loaded when the timer expires. The boot options can be enumerated by the UEFI boot manager by reading the *BootOrder* and *Boot###* environment variables. These environment variables are discussed in sections 3.1 and 3.2 of the *UEFI 2.0 Specification*. A boot option is typically an OS loader that never returns to EFI, but boot options can also be UEFI applications like diagnostic utilities or the UEFI Shell. If a boot option does return to the UEFI boot manager, and the return status is not **EFI_SUCCESS**, then the UEFI boot manager the next boot option will be processed. This process is repeated until an OS is booted, **EFI_SUCESS** is returned by a boot option or the list of boot options is exhausted. Once the boot process has halted, the UEFI boot manager will typically provide a user interface that allows the user to manually boot an OS or manage the platform.

The UEFI boot manager will use the device path in each boot option to ensure that the device required to access the boot option has been added to the UEFI handle database. This process is exactly the same as the process used for the console variables *ErrOut*, *ConOut*, and *ConIn*.

3 Coding Conventions

This section describes the coding conventions that are used in the *EFI Developer's Kit and most of the open source code.* Most of the code that is distributed in the *EFI Developer's Kit* is implemented in the C programming language. One goal of the UEFI Driver Model is to allow portable UEFI drivers to be designed such that they can be easily compiled for various processor architectures. The instruction set architectures supported today include the following:

- IA-32
- Intel-64 (64 bit extensions)
- IA-64
- EFI Byte Code (EBC)

The EFI Byte Code virtual machine architecture was optimized with the C programming language in mind, so if an UEFI driver writer wishes to design for portability, then the UEFI driver must be implemented in the C programming language with no use of assembly language. As a result, the coding convention presented here concentrates on conventions for the C programming language. This coding convention is also used in the code examples presented throughout this document.

UEFI drivers are not required to use these coding conventions. There is no data that suggests that one code convention is superior to another. However, there is a large body of data that suggests that the use of a consistent coding convention improves the efficiency of software development. There are many advantages to using a common coding convention, including the following:

- Source code is easier to read.
- Source code is easier to maintain.
- It improves collaboration between developers.
- UEFI driver are easier to debug.

Recommendations are presented for the naming of files, functions, macros, data types, variables, and constants. A consistent naming convention helps improve the readability of the code and it tends to make the code somewhat self documenting, which may help reduce the required comment overhead. In addition, a consistent function naming convention improves the readability of call stacks during debug. Conventions for the use of tabs, spaces, indentation, and comments are also presented along with templates for the various expressions and constructs that are available in the C programming language. Finally, the special considerations for implementing new UEFI protocols, UEFI GUIDs, and UEFI drivers are presented.

3.1 Indentation and Line Length

Tabs are not used in any of the source files. Instead, spaces are used for all indentation. All code blocks are indented in increments of two spaces. This two-space indentation is smaller than some other conventions that use four spaces, but this reduced indentation allows more code to be viewed on a single screen even when fairly deep nesting is used.

In general, the length of a line of code does not exceed 80 characters. This length is only a guideline, however. The main reason for defining a guideline on line length is so code can be easily viewed in a full-screen editor and formatted correctly when it is sent to a printer.

3.2 Comments

3.2.1 File Headers Comments

Every **.c** and **.h** file has a file header comment block at the very top of the file. The file header comment block has the form shown in Example 3.

```
/*++

Copyright (c) 2003 Xyz Corporation

Module Name:

  <<file name, e.g. Foo.c>>

Abstract:

  <<description of file contents>>

--*/
```

Example 3. File Header Comment Block for .C and .H Files

Files with an extension of **.inf** are essentially a shorthand **makefile** that the build tools in the *EDK* use to determine the source files and libraries that are required to build an UEFI driver. Every **.inf** file has a file header comment block at the very top of the file as shown in Example 4.

```
#
# Copyright (c) 2003 Xyz Corporation
#
# Module Name:
#
#   <<file name, e.g. Make.inf>>
#
# Abstract:
#
#   <<description of the file contents, e.g. Makefile for
EDK\Drivers\DiskIo>>
#
```

Example 4. File Header Comment Block for .INF Files

3.2.2 Function Header Comments

Each function has a comment block between the parameter declaration section of the function definition and the opening brace of the function body. The only exceptions are simple or small private functions. The function header comment block takes the form as shown in Example 5.

```
EFI_STATUS
Foo (
  IN UINTN  Arg1,
  IN UINTN  Arg2
  )
/*++

Routine Description:

  <<description>>

Arguments:

  <<argument names and purposes>>

Returns:

  <<description of possible return values>>

--*/
{
}
```

Example 5. Function Header Comment Block

3.2.3 Internal Comments

Internal code comments use C++ style (//) comment lines. A block of comment lines that contain text are preceded and followed by a blank comment line. A blank line may optionally follow a comment block. The blank line generally indicates that the comment is for a large block of code. If a comment block is not followed by a blank line, then the comment only applies to the next few lines of code. The comments are indented with the code.

Comments are not placed on the same line as source code, but comments may be on the same line as data structures declarations and data structure initializations. Commenting is done as efficiently as possible because too many comments are as bad as too few. Comments are added to explain why things are done and the big picture of how a module works. Example 6 shows examples of internal comments.

```
//
// This is an example comment for the large block of code below
//

if (Test) {
  //
  // This is an example comment for the next code line
  //
  return Test;
}
```

Example 6. Internal Comments

3.2.4 What Is Commented

The following things are commented in source code:

- Complicated, tricky, or sensitive pieces of code. Making the code cleaner is often better than adding more comments.
- Higher-level concepts in the code. Focus on the why and not the how.
- Data structures and `#define` statements in include files.
- The version number of any industry standard that is referenced by the code.

3.2.5 What Is Not Commented

The following things are not commented in source code:

- In general, a single line of code does not need a comment.
- Where possible, the code is self-documenting. Do not repeat the code or explain it in a comment. Instead, comments clarify the intent of the code or explain higher-level concepts.
- Do not place markers in the code, such as a developer's name or unusual patterns. They may have meaning to the developer but do not to other developers or projects.
- Code is not removed or disabled by using comments. Instead, a source control system is used to maintain different source code file revisions.

3.3 General Naming Conventions

Good naming practice is used when declaring variable names. Studies show that programs with names that average 10 to 16 characters in length are the easiest to debug (*Code Complete*). Programs with names averaging 8 to 20 characters are almost as easy to debug. This name length is simply a guideline. The most important thing is that the name conveys a clear meaning of what it represents.

Names are formed with a mix of upper and lower case text. Each word or concept starts with a capital letter and is followed by lower case letters for the rest of the word. Only the first letter of an acronym is capitalized. For example:
`ThisIsAnExampleOfWhatToDoForPci`

Commonly used terms are not be overloaded when possible. For example, UEFI has an event model, so creating a new abstraction called an “event” would have caused confusion. The goal is to use names such that other developers with very little context can understand what each name represents.

3.3.1 Abbreviations

Table 11 describes a common set of abbreviations. Abbreviations do not have to be used, but if abbreviations are used, then the common one is selected. If new abbreviations are required, then making the abbreviation easy to remember is more important to the reader of the code than it is to the writer of the code. New abbreviations are declared in a comment block at the top of the file. The translation table contains the new abbreviations and definitions. An attempt is made not to define new abbreviations to replace an existing abbreviation. For example, *No* is not redefined to mean *Number*, because *Num* is already defined as the supported abbreviation.

Table 11. Common Abbreviations

Abbreviation	Description
Ptr	Pointer
Str	Unicode string
Ascii	ASCII string
Len	Length
Arg	Argument
Max	Maximum
Min	Minimum
Char	Character
Num	Number
Temp	Temporary
Src	Source
Dst	Destination
BS	EFI Boot Services Table
RT	EFI Runtime Table
ST	EFI System Table
Tpl	EFI Task Priority Level

3.3.2 Acronyms

The use of acronyms is limited. The idea is to code for a developer who will have to read and maintain the code. Making up a new vernacular to describe a module can lead to considerable confusion. If new acronyms are created, then they are fully defined in the documentation and each module that uses the acronym contains a comment with a translation table in the file header.

The use of acronyms for industry standards is acceptable. Acronyms such as EFI, PCI, ACPI, SMBios, and USB (capitalized to the variable naming convention) are used without defining their meaning. If an industry standard acronym is referenced, then the file header defines which revision of the specification is being assumed. For example, a PCI resource manager would state that it was coded to follow the PCI 2.2 specification.

3.4 Directory and File Names

Directory names and file names follow the general naming conventions that were presented in the previous section. There are no artificial limits on the length of a directory name or a source file name. None of the build tools in the *EDK* require the file names to follow the 8.3 naming convention. New code is added below the **EDK** directory in the *EDK*. Example 7 contains some example directory names and file names from the area of the source tree where new drivers, protocols, GUIDs, and libraries would be placed during development. This example shows the source files for the following:

- The PCI video driver that produces the **EFI_GRAPHICS_OUTPUT_PROTOCOL**
- The declaration of the **EFI_BLOCK_IO_PROTOCOL**
- **EFI_GLOBAL_VARIABLE_GUID**, which is used to access UEFI variables
- A few source files from the UEFI driver Library

It is important for these directory and file name conventions to be followed when directories and files are created and when directory or file names are referenced from source files. These conventions will provide compatibility with case-sensitive file systems.

```
EDK\
  Sample\
    BUS\
      PCI\
        CirrusLogic\
          DXE\
            UEFICirrusLogic5430GraphicsOutput.c
            UEFICirrusLogic5430GraphicsOutput.h
  Foundation\
    Efi\
      Protocol\
        BlockIO
          BlockIO.c
          BlockIO.h
      Guid\
        GlobalVariable.c
        GlobalVariable.h
  Sample\
    Bus\
      PCI\
        Undi\
          RuntimeDxe\
            E100B.c
            E100B.h
```

Example 7. Directory and File Names

3.5 Function Names and Variable Names

Data variables and function names follow the general naming conventions. Each word or concept starts with a capital letter and is followed by lowercase letters. Macros, **#define**, and **typedef** declarations are all capital letters. Each word is separated by the underscore '_' character. The use of '**_t**' or '**_T**' is discouraged. Example 8 shows some example function and variable names.

```
#define THIS_IS_AN_EXAMPLE_OF_WHAT_TO_DO_FOR_PCI 1

#define MY_MACRO(a) ((a) == 1)

typedef struct {
  UINT32  Age;
  CHAR16  Name[32];
} MY_STRUCTURE;

EFI_STATUS
GetStructure (
  MY_STRUCTURE  *MyStructure
)
{
  EFI_STATUS  Status;
  . . .
}
```

Example 8. Function and Variable Names

3.5.1 Hungarian Prefixes

The Hungarian naming convention is generally discouraged in the coding conventions described here except for the global variable. The Hungarian naming convention is a set of detailed guidelines for naming variables and routines. The convention is widely used with the C programming language. The term "Hungarian" refers both to the fact that the names that follow the convention look like words from a foreign language and that the creator of the convention, Charles Simoyi, is originally from Hungary. The following is an example of a variable named with the "Hungarian" conventions:

pachInsert - A pointer to an array of characters to insert.

Hungarian conventions are not recommended for the following reasons:

The abstraction of abstract data types is not covered.

Instead, base types based on C programming language type–like integers and long integers are abstracted. As a result, the names are focused on data types instead of the object-oriented abstractions that they represent. This focus is of little value and forces manual type checking that can just as easily be accomplished by using a compiler with strong type checking or other source code analysis tools.

Hungarian combines data meaning with data representation.

If a change is made to a data type, then all the variables of that data type have to be renamed. In addition, there is no mechanism to ensure that the names are accurate.

Studies have shown that Hungarian notation tends to encourage lazy variable names (*Code Complete*).

It is common for developers to focus on the Hungarian prefix without putting effort into a good descriptive name.

3.5.2 Global Variables and Module Variables

The only exception to the Hungarian prefix model is the prefixing of a global variable with a 'g' and a module variable with an 'm.' For example:

```
gThisIsAGlobalVariableName
mThisIsAModuleVariableName
```

The use of global data is discouraged by this coding convention. The use of module variables is appropriate for solving specific programming issues. A *module* is defined to be a set of data and routines that act on that data. Thus, an UEFI driver that produces a protocol can be thought of as a module. A complex protocol may be built out of several smaller modules. Any variable with scope outside of a single routine is prefixed by an 'm' or a 'g.'

Some module variables and global variables will require a locking mechanism to guarantee that the variable is being accessed by only one agent at a time. The locking mechanism that is available to UEFI drivers is discussed in detail in chapter 4 of this document.

The difference between a module variable and a global variable is not always obvious. A module variable is accessed only by a small set of routines, typically within a single source file, that have strict rules for accessing the module variable. A global variable is accessed throughout the program, typically from multiple source files. Accessing a module variable would not be common over the life cycle of a program. New code, bug fixes, or new features would access the current routines that are used to abstract the module variable. The module variable would be easy to change, because it is accessed only from a small number of routines within a single source file. On the other hand, a global variable is accessed throughout the UEFI driver and as the UEFI driver evolves, more code will tend to access the global variable. Global variables and module variables are valid if they are being used to implement good information that is hiding in the design.

3.6 Macro Names

Macros use a different naming convention than functions. The main reason is the difference in the order of precedence that can occur between poorly constructed macros and functions. An overuse of parentheses is strongly encouraged, because it is very difficult to debug a precedence order bug in a macro. The following are examples of macro construction:

```
#define BAD_MACRO(a, b) a*b
#define GOOD_MACRO(a, b) ((a)*(b))
```

The following examples show the difference between **`BAD_MACRO()`** and **`GOOD_MACRO()`**:

- **`BAD_MACRO (10, 2)`** and **`GOOD_MACRO (10, 2)`** both evaluate to 20.
- **`BAD_MACRO (7+3, 2)`** evaluates to 7 + 3 * 2 = 7 + (3 * 2) = 7 + 6 = 13.

- **GOOD_MACRO (7+3, 2)** evaluates to (7+3) * (2) = 10 * 2 = 20.

3.6.1 Macros as Functions

It is recommended that all macros be constructed using the macro rules. Even with these recommendations, it is possible for a macro and an equivalent function to behave differently. All macros are designed so that they can be easily converted to an equivalent function implementation.

Example 9 shows both good and bad examples of macros and their equivalent function implementations.

```
#define BAD_MACRO(Value)   ((Value) == 1) || ((Value) == 2)

#define GOOD_MACRO(Value)   ((Value) == 1)

BOOLEAN
BadMacro (
  UINTN Value
)
{
  return ((Value == 1) || (Value == 2));
}

BOOLEAN
GoodMacro (
  UINTN  Value
)
{
  return (Value == 1);
}
```

Example 9. Macros as Functions

When **GOOD_MACRO(Value)** and **GoodMacro(Value)** are called, the results are identical.

When **BAD_MACRO(Value)** and **BadMacro(Value)** are called, the results are identical.

When **GOOD_MACRO(Value++)** and **GoodMacro(Value++)** are called, the results are identical.

When **BAD_MACRO(Value++)** and **BadMacro(Value++)** are called, the results are not identical when *Value* is zero. Also, when **BAD_MACRO(Value++)** is called, *Value* is incremented twice, and when **BadMacro(Value++)** is called, *Value* is incremented only once.

3.7 Data Types

The UEFI coding convention defines a set of common data types that are used to ensure portability between different compilers and processor architectures. Any abstract type that is defined is constructed from other abstract types or from common UEFI data types. The types **int**, **unsigned**, **char**, **void**, **static**, and **long** are not used in this coding convention.

There is one exception to the preceding rules. The use of **static** is used for data but not for functions. The **STATIC** keyword is used to disable the static nature of function names to make their scope global to aid in debugging. This conversion does not influence the behavior of the program. Converting a **static** data type to non-static types can impact the way a program functions and thus the use of **static** for data is allowed.

Table 12 contains common data types that are referenced in the interface definitions defined in the *UEFI 2.0 Specification*. Unless otherwise specified, all data types are naturally aligned. Structures are aligned on boundaries that are equal to the largest internal datum of the structure, and internal data is implicitly padded to achieve natural alignment.

The mappings from these compiler-dependent types to the UEFI base types are handled in a single include file called **EfiBind.h**. If a new compiler is used, then **EfiBind.h** may need to be updated, but once the mapping between the new compiler's types and the UEFI base types is made, all existing UEFI source code should compile correctly.

Table 12. Common UEFI Data Types

Mnemonic	Description
BOOLEAN	Logical Boolean. 1-byte value containing a 0 for **FALSE** or a 1 for **TRUE**. Other values are undefined.
INTN	Signed value of native width (4 bytes on IA-32 and 8 bytes on IA-64 instruction set architecture operations).
UINTN	Unsigned value of native width (4 bytes on IA-32 and 8 bytes on IA-64 instruction set architecture operations).
INT8	1-byte signed value.
UINT8	1-byte unsigned value.
INT16	2-byte signed value.
UINT16	2-byte unsigned value.
INT32	4-byte signed value.
UINT32	4-byte unsigned value.
INT64	8-byte signed value.
UINT64	8-byte unsigned value.
CHAR8	1-byte character.
CHAR16	2-byte character. Unless otherwise specified, all strings are stored in the UTF-16 encoding format as defined by Unicode 2.1 and ISO/IEC 10646 standards.
VOID	Undeclared type.
EFI_GUID	128-bit buffer containing a unique identifier value. Unless otherwise specified, aligned on a 64-bit boundary.
EFI_STATUS	Status code. Type INTN.
EFI_HANDLE	A collection of related interfaces. Type VOID *.
EFI_EVENT	Handle to an event structure. Type VOID *.

EFI_LBA	Logical block address. Type UINT64.
EFI_TPL	Task priority level. Type UINTN.
EFI_MAC_ADDRESS	32-byte buffer containing a network Media Access Controller address.
EFI_IPv4_ADDRESS	4-byte buffer. An IPv4 internet protocol address.
EFI_IPv6_ADDRESS	16-byte buffer. An IPv6 internet protocol address.
EFI_IP_ADDRESS	16-byte buffer aligned on a 4-byte boundary. An IPv4 or IPv6 internet protocol address.
<Enumerated Type>	Element of an enumeration. Type INTN.

Table 13 defines modifiers that are used in function and data declarations. The **IN**, **OUT**, and **OPTIONAL** modifiers are used only to qualify arguments to functions and therefore do not appear in data type declarations. The **IN** and **OUT** modifiers are placed at the beginning of the line before the data type of each function argument, and the **OPTIONAL** modifier is placed at the end of the line after the name of the function argument. The **STATIC** modifier is used to modify the scope of a function and can be overloaded to support debugging. The **EFIAPI** modifier is used to ensure that the correct calling convention that is defined in the *UEFI 2.0 Specification* is used between different modules that are not linked together.

Table 13. Modifiers for Common UEFI Data Types

Mnemonic	Description
IN	Datum is passed to the function. Placed at the beginning of a source line before the data type of the function argument.
OUT	Datum is returned from the function. Placed at the beginning of a source line before the data type of the function argument.
OPTIONAL	Datum that is passed to the function is optional, and a **NULL** may be passed if the value is not supplied. Placed at the end of a source line after the name of the function argument.
STATIC	The function has local scope. This modifier replaces the standard C static key word, so it can be overloaded for debugging.
VOLATILE	Declares a variable to be volatile and thus exempt from optimization to remove redundant or unneeded accesses. Any variable that represents a hardware device is declared as **VOLATILE**.
CONST	Declares a variable to be of type **const**. This modifier is a hint to the compiler to enable optimization and stronger type checking during compile time.
EFIAPI	Defines the calling convention for UEFI interfaces. All UEFI intrinsic services and any member function of a protocol use this modifier on the function definition.

3.7.1 Enumerations

Example 10 is an example of a construction for an enumerated type. The names of the elements in the enumerated type follow the same naming convention as variables and functions. The **enum** is declared as a **typedef** with the name of the **typedef** following the standard all-capitalization rules. It is recommended that the last element of the **enum** be a member element that represents the maximum legal value for the enumeration. This representation allows for bounds checking on an **enum** to support debugging and sanity checking the value assigned to an **enum**. It is also recommended that the **enum** members be named carefully so that their names do not collide with other variable or function names.

```
typedef enum {
  EnumMemberOne,
  EnumMemberTwo,
  EnumMemberMax
} ENUMERATED_TYPE;
```

Example 10. Enumerated Types

3.7.2 Data Structures and Unions

Example 11 is an example of a construction of a data structure and a union. The names of the field in the data structure and the union follow the same naming conventions as variable and functions. A **struct** or **union** is declared as a **typedef** with the name of the **typedef** following the standard all-capitalization rules.

```
typedef struct {
  UINT32  FieldOne;
  UINT32  FieldTwo;
  UINT32  FieldThree;
} MY_STRUCTURE;

typedef union {
  UINT16  Integer;
  CHAR16  Character;
} MY_UNION;
```

Example 11. Data Structure and Union Types

3.8 Constants

Table 14 lists the constants that are available to all UEFI drivers.

Table 14. UEFI Constants

Mnemonic	Description
TRUE	One
FALSE	Zero
NULL	**VOID** pointer to zero.

3.9 Include Files

Include files contain a **#ifndef** and a **#define** statement at the start of the include file and a **#endif** statement as the last line of the file. The **#ifndef** and **#define** statements contain an all-uppercase version of the include file name, prefixed and suffixed by the '_' character, which prevents duplicate definitions if the same include file is included by a different module. All C include files use the **.h** file extension. The example in Example 12 shows the include file **SerialDriver.h**. It contains **#ifndef** and **#define** statements of **_EFI_SERIAL_DRIVER_H_**.

```
/*++

Copyright (c) 2003 Xyz Corporation

Module Name:

  SerialDriver.h

Abstract:

  Serial driver ….

--*/

#ifndef _EFI_SERIAL_DRIVER_H_
#define _EFI_SERIAL_DRIVER_H_

//
// Statements that include other header files
//

//
// Simple defines of such items as status codes and macros
//

//
// Type declarations
//

//
// Function prototype declarations
//

#endif
```

Example 12. Include File

The above comments suggest an order of declarations in an include file. Include files contain public declarations or private declarations, but not both. Include files do not contain code or declare storage for variables. Examples of public include files would be protocol definitions or industry standard specifications (such as EFI, ACPI, and SMBIOS). Private include files would contain static functions and internal data structure definitions.

3.10 Spaces in C Code

Spacing guidelines are very important and greatly improve the readability of source code. The program shown in Example 13 is an example that uses poor spacing guidelines and is very difficult to follow (Dishonorable mention, Obfuscated C Code Contest, 1984. Author requested anonymity).

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-
',i+++"hello,world!\n",'/'/'/' ));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Example 13. Poor Spacing

3.10.1 Vertical Spacing

Blank lines are used to make code more readable and to group logically related sections together. The convention of one statement per line is followed. This convention is not followed in **for** loops, where the initial, conditional, and loop statements reside on a single source line. The **if** statement is not an exception to this guideline. The conditional expression and body go on separate lines. The open brace (**{**) is placed at the end of the first line of the construct. The close brace (**}**) and **case** labels are placed on their own line. The close brace does share a line with the **else** and **else if** constructs.

3.10.2 Horizontal Spacing

Spaces are placed around assignment operators, binary operators, after commas, and before an open parenthesis. Spaces are not placed around structure members, pointer operators, or before open brackets of array subscripts. It is also better to use extra parentheses in more complex expressions rather than to depend on in-depth knowledge of the precedence rules of C. In addition, continuation lines are formatted so that they line up with the portion of the preceding line that they continue. Example 14 contains several practical examples of these spacing conventions.

```
Status = TestString (String, Index + 3, &Value);
MinResult = MIN_EXAMPLE (Start, End);
Data.Index
Pointer->Index = *Ptr;
Array[(Max + Min) / 2]
Value = (GetData (BubbaBaes + BUBBA_HIGH_DATA) << 8) |
         GetData (BubbaBaes + BUBBA_LOW_DATA)
```

Example 14. Horizontal Spacing Examples

3.11 Standard C Constructs

The following sections describe the code conventions for the following C constructs:

- Subroutines
- Function calls
- Boolean expressions
- Conditional expressions
- Loop expressions
- Switch expressions

3.11.1 Subroutines

Function prototype declarations and function implementations use the same coding conventions. The first line is the data type of the return value, left justified. If the function is a protocol member, then the next line is **EFIAPI**. The next line is the name of the function that follows the function naming conventions, followed by a space and an open parenthesis. This line is followed by the argument list with one argument per line, indented by two spaces. The type names for each argument are aligned on the same column, and the beginning of the field name for each argument is also aligned in the same column. The last argument is followed by a line with a close parenthesis that is also indented by two spaces.

Each argument type definition is preceded by an **IN** and/or **OUT** modifiers. The modifiers are used to indicate whether the argument is an input or output variable. The **IN** variables are listed first followed by the **OUT** variables. If data is both passed in and passed out through a variable, then it is marked as both **IN** and **OUT**. A buffer that is passed into a routine that modifies the contents of the buffer is marked as both **IN** and **OUT**. Table 15 shows the code conventions that are used for **IN** and **OUT**.

Table 15. IN and OUT Usage

Mnemonic	Description
IN	Passed by value. For C, this mnemonic is any argument whose name is not preceded by an asterisk (*). Placed at the beginning of a line containing a function argument.
IN	Passed by reference, and referenced data is not modified by the routine. An asterisk precedes the argument name. Placed at the beginning of a line containing a function argument.
OUT	Passed by reference, and the referenced data is modified by the routine. The passed-in state of the referenced data is not used by the routine. Placed at the beginning of a line containing a function argument.
IN OUT	Passed by reference, and the passed-in referenced data is consumed and then modified by the routine. Placed at the beginning of a line containing a function argument.
OPTIONAL	If argument is a **NULL** pointer, it is not present. If the value is not **NULL**, it is a valid argument. Placed at the end of a line containing a function argument.

Function implementations are then followed by a function comment header and the function body. The function body starts with the declaration of all the local variables that are used in the function. Local variable declarations may not be spread throughout the function. There is one local variable declared per line, indented by two spaces. Like the argument declarations, local variable names are aligned in the same column. Local variables declarations are not commented. Instead, they have self-describing names. If comments are required for complex local variable declarations, then the comments are placed in the include file that defines the complex data type, or the comments are placed in the function's comment header. Local variables are not initialized as part of their declaration. Instead, the local variables are initialized as part of the code body that follows the local variable declarations. Example 15 shows an example of the function prototype and function implementation.

```
//
// Function Prototype Declaration
//
EFI_STATUS
EFIAPI
FooName (
  IN     UINTN  Arg1,
  IN     UINTN  Arg2, OPTIONAL
  OUT    UINTN  *Arg3,
  IN OUT UINTN  *Arg4
  );

//
// Function Implementaion
//
EFI_STATUS
EFIAPI
FooName (
  IN     UINTN  Arg1,
  IN     UINTN  Arg2, OPTIONAL
  OUT    UINTN  *Arg3,
  IN OUT UINTN  *Arg4
  )
/*++

Routine Description:

  <<description>>

Arguments:

  <<argument names and purposes>>

Returns:

  <<description of possible return values>>

--*/
{
  UINTN  LocalOne;
  UINTN  LocalTwo;
  UINTN  LocalThree;
  . . .
}
```

Example 15. C Subroutine

3.11.2 Calling Functions

Function calls contain the function name, followed by a space and an open parenthesis. If all the arguments fit on one line, then the arguments are separated by commas on that line. If the arguments do not all fit on the same line, then each argument is placed on its own line, indented two spaces from the first character of the function being called. Example 16 shows the code conventions for calling functions.

```
//
// Function and arguments fit on a single line
//
Foo (A, B, C);

//
// Function and arguments do not fit on a single line
//
Status = BlockIo->ReadBlocks (
                    BlockIo,
                    MediaId,
                    Lba,
                    BufferSize,
                    Buffer
                    );
```

Example 16. Calling Functions

3.11.3 Boolean Expressions

Boolean tests are constructed using the following conventions. Boolean values and variables of type **BOOLEAN** do not require explicit comparisons to **TRUE** or **FALSE**. Non-Boolean values or variables use comparison operators (**==**, **!=**, **>**, **<**, **>=**, **<=**). A comparison of any pointer to zero is done with the **NULL** constant. Example 17 shows an example of the code conventions for Boolean expressions.

```
BOOLEAN  Done;
UINTN    Index;
VOID     *Ptr;

//
//          Incorrect                       Correct
//   ======================          =========================
     if (Index) {                    if (Index != 0) {
     if (!Index) {                   if (Index == 0) {
     if (Done == TRUE) {             if (Done) {
     if (Done == FALSE) {            if (!Done) {
     if (Ptr) {                      if (Ptr != NULL) {
     if (Ptr == 0) {                 if (Ptr == NULL) {
```

Example 17. Boolean Expressions

3.11.4 Conditional Expressions with examples

Example 18 contains examples of the coding conventions for condition expressions, including the following:

3.11.4.1 An “if” construct

```
//
// IF construct
//
if (A > B) {
  IamTheCode ();
}
```

3.11.4.2 An “if/else” construct

```
//
// IF / ELSE construct
//
if (Pointer == NULL) {
  IamTheCode ();
} else {
  IamTheCode ();
}
```

3.11.4.3 An “if/else if/else” construct

```
//
// IF / ELSE IF / ELSE construct
//
if (Done == TRUE) {
  IamTheCode();
} else if (A < B) {
  IamTheCode();
} else {
  IamTheCode();
}
```

3.11.4.4 A nested “if” construct

```
//
// Nested IF construct
//
if (A > 10) {
  if (A > 20) {
    IamTheCode ();
  } else {
    if (A == 15) {
      IamTheCode ();
    } else {
      IamTheCode ();
    }
  }
}
```

The **if** statement is followed by an open brace (**{**) even if the body contains only one statement. The body is indented two spaces, and the close brace (**}**) is placed on its own line and shares the same indentation as the **if** statement. The close brace (**}**) may optionally be followed by an **else if** or an **else** statement and an additional open brace (**{**). The body for the **else if** and the **else** statements are also indented by two spaces.

3.11.5 Loop Expressions with examples

3.11.5.1 A while loop

```
//
// WHILE Loop construct
//
while (Pointer != NULL) {
  IamTheCode();
}
```

3.11.5.2 A do loop

```
//
// DO Loop construct
//
do {
  IamTheCode();
} while (A < B);
```

3.11.5.3 A for loop

```
//
// FOR Loop construct
//
for (Index = 0; Index < MAX_INDEX; Index++) {
  IamTheCode(Index);
}
```

3.11.5.4 A nested for loop

```
//
// Nested Loop construct
//
for (Column = 0; Column < MAX_COLUMN; Column++) {
  for (Ror = 0; Row < MAX_ROW; Row++) {
    IamTheCode(Index);
  }
}
```

The loop statements are followed by an open brace (**{**) on the same line even if the body contains only one statement. The body is indented two spaces, and the close brace (**}**) is placed on its own line and shares the same indentation as the loop statement.

3.11.6 Switch Expressions

Example 18 contains an example of a **switch** statement. The **switch** statement is followed by an open brace (**{**) on the same line. The **case** statements and **default** statements are placed on their own lines at the same indentation level as the **switch** statement. The body of the **case** statements and **default** statements are indented two spaces, and the close brace (**}**) is placed on its own line and shares the same indentation as the **switch** statement.

```
switch (Variable) {

case 1:
  IamTheCode ();
  break;

case 2:
  IamTheCode ();
  break;

default:
  IamTheCode ();
  break;
}
```

Example 18. Switch Expressions

3.11.7 Goto Expressions

In general, the **goto** construct is not used. However, it is acceptable to use **goto** for error handling and thus exiting a routine in an error case. A **goto** allows the error exit code to be contained in one place at the end of a routine. This location reduces software life cycle maintenance issues, as there can be one copy of error cleanup code per routine. If a **goto** is not used for this case, then the error cleanup code tends to get replicated multiple times, which tends to induce errors in code maintenance. The **goto** follows the normal rules for C code. The labels are left justified. Example 19 shows the code convention for **goto** expressions.

```
{
  EFI_STATUS  Status;

  . . .
  Status = IAmTheCode ();
  if (EFI_ERROR (Status)) {
    goto ErrorExit;
  }

  IDoTheWork ();

ErrorExit:
  . . .
  return Status;
}
```

Example 19. Goto Expressions

3.12 EFI File Templates

This section contains templates and guidelines for creating files for UEFI protocols, UEFI GUIDs, and UEFI drivers. The naming conventions for the driver entry point and the functions exported by a driver that are presented here will guarantee that a unique name is produced for every function, which aides in call stack analysis when root-causing driver issues. The following expressions are used throughout this section to show where protocol names, GUID names, function names, and driver names should be substituted in a file template:

<<ProtocolName>>

Represents the name of a protocol that follows the function or variable naming convention, which capitalizes only the first letter of each word (e.g., **DiskIo**).

<<PROTOCOL_NAME>>

Represents the name of a protocol that follows the data structure naming convention, which capitalizes all the letters and separates each word with an underscore '_' (e.g., **DISK_IO**).

<<FunctionNameN>>

Represents the *nth* name of the protocol member functions that follow the function or variable naming convention, which capitalizes only the first letter of each word (e.g., **ReadDisk**).

<<FUNCTION_NAMEn>>

Represents the *nth* name of the protocol member functions that follows the data structure naming convention, which capitalizes all the letters and separates each word with an underscore '_' (e.g., **READ_DISK**).

<<GuidName>>

Represents the name of a GUID that follows the function or variable naming convention, which capitalizes only the first letter of each word (e.g., **GlobalVariable**).

<<GUID_NAME>>

Represents the name of a GUID that follows the data structure naming convention, which capitalizes all the letters and separates each word with an underscore '_' (e.g., **GLOBAL_VARIABLE**).

<<DriverName>>

Represents the name of a driver that follows the function or variable naming convention, which capitalizes only the first letter of each word (e.g., **Ps2Keyboard**).

<<DRIVER_NAME>>

Represents the name of a driver that follows the data structure naming convention, which capitalizes all the letters and separates each word with an underscore '_' (e.g., **PS2_KEYBOARD**).

<<DriverVersion>>

Value that represents the version of the driver. Values from 0x0–0x0f and 0xFFFFfff0–0xFFFFFFFF are reserved for UEFI drivers that are written by OEMs for integrated devices. Values from 0x10–0xFFFFffef are reserved for UEFI drivers that are written by IHVs.

<<ProtocolNameCn>>

Represents the *nth* name of a protocol that is consumed by an UEFI driver and follows the function or variable naming convention, which capitalizes only the first letter of each word (e.g., **DiskIo**).

<<PROTOCOL_NAME_CN>>

Represents the *nth* name of a protocol that is consumed by an UEFI driver and follows the data structure naming convention, which capitalizes all the letters and separates each word with an underscore '_' (e.g., **DISK_IO**).

<<ProtocolNamePm>>

Represents the *mth* name of a protocol that is produced by an UEFI driver and follows the function or variable naming convention, which capitalizes only the first letter of each word (e.g., **DiskIo**).

<<PROTOCOL_NAME_PM>>

Represents the *mth* name of a protocol that is produced by an UEFI driver and follows the data structure naming convention, which capitalizes all the letters and separates each word with an underscore '_' (e.g., **DISK_IO**).

3.12.1 Protocol File Templates

Protocols are placed in the *EDK* subdirectory **\Foundation\Protocol\<<ProtocolName>>**.

These directories contain a **<<ProtocolName>>.c** and a **<<ProtocolName>>.h** file. The **<<ProtocolName>>.c** file has the contents shown in Example 20. The file header comments have also been removed from this template. The **<<ProtocolName>>.h** file has the contents shown in Example 21. The file header comments and function header comments have been omitted from this template. The GUID shown is an illegal GUID that is composed of all zeros. Every new protocol must generate a new GUID. **GUIDGEN**, which is shipped with Microsoft Visual Studio, can be used to generate new GUIDs. The full source to the Disk I/O Protocol is included in Appendix D for reference.

```
#include "Efi.h"
#include EFI_PROTOCOL_DEFINITION (<<ProtocolName>>)

EFI_GUID gEfi<<ProtocolName>>ProtocolGuid =
EFI_<<PROTOCOL_NAME>>_PROTOCOL_GUID;

EFI_GUID_STRING (&gEfi<<ProtocolName>>ProtocolGuid, "ShortString",
"LongString");
```

Example 20. Protocol C File

```
#ifndef __EFI_<<PROTOCOL_NAME>>_H__
#define __EFI_<<PROTOCOL_NAME>>_H__

#define EFI_<<PROTOCOL_NAME>>_PROTOCOL_GUID \
  { 0x00000000, 0x0000, 0x0000, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00 }

EFI_INTERFACE_DECL (_EFI_<<PROTOCOL_NAME>>_PROTOCOL);

typedef
EFI_STATUS
(EFIAPI *EFI_<PROTOCOL_NAME>>_<<PROTOCOL_FUNCTION_NAME1>>) (
  IN EFI_<<PROTOCOL_NAME>>_PROTOCOL *This,
  //
  // Place additional function arguments here.
  //
  );

typedef
EFI_STATUS
(EFIAPI *EFI_<PROTOCOL_NAME>>_<<PROTOCOL_FUNCTION_NAME2>>) (
  IN EFI_<<PROTOCOL_NAME>>_PROTOCOL *This,
  //
  // Place additional function arguments here.
  //
  );

// . . .

typedef
EFI_STATUS
(EFIAPI *EFI_<PROTOCOL_NAME>>_<<PROTOCOL_FUNCTION_NAMEn>>) (
  IN EFI_<<PROTOCOL_NAME>>_PROTOCOL *This,
  //
  // Place additional function arguments here.
  //
  );

typedef struct _EFI_<<PROTOCOL_NAME>>_PROTOCOL {
  EFI_<<PROTOCOL_NAME>>_<<FUNCTION_NAME1>>  <<FunctionName1>>;
  EFI_<<PROTOCOL_NAME>>_<<FUNCTION_NAME2>>  <<FunctionName2>>;
  // . . .
  EFI_<<PROTOCOL_NAME>>_<<FUNCTION_NAMEn>>  <<FunctionNameN>>;
  //
  // Place protocol data fields here
  //
} EFI_<<PROTOCOL_NAME>>_PROTOCOL;

extern EFI_GUID gEfi<<ProtocolName>>ProtocolGuid;

#endif
```

Example 21. Protocol Include File

3.12.2 GUID File Templates

GUIDs and their associated data structures are declared just like protocols. The only difference is that GUIDs are placed in the *EDK* subdirectory **\Foundation\Guid\<<GuidName>>**. These directories contain a **<<GuidName>>.c** and a **<<GuidName>>.h** file. The **<<GuidName>>.h** file has the contents shown in Example 22. The file header comments have been omitted from this template. The GUID shown is an illegal GUID that is composed of all zeros. The **GUIDGEN** tool can be used to generate new GUIDs. The **<<GuidName>>.c** file has the contents shown in Example 23. The file header comments have also been omitted from this template. The full source to the UEFI global variable GUID is included in Appendix D for reference. GUIDs are added to the **\Foundation\Guid\<<GuidName>>** *EDK* subdirectory when a GUID is required by more than one UEFI component. If a GUID is required only by a single UEFI driver, then it can be declared with the source code to the UEFI driver.

```
#ifndef __<<GUID_NAME>>_H__
#define __<<GUID_NAME>>_H__

#define EFI_<<GUID_NAME>>_GUID \
  { 0x00000000, 0x0000, 0x0000, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00 }

typedef struct {
  //
  // Place GUID specific data fields here
  //
} EFI_<<GUID_NAME>>_GUID;

extern EFI_GUID gEfi<<GuidName>>Guid;

#endif
```

Example 22. GUID Include File

```
#include "Efi.h"
#include EFI_GUID_DEFINITION (<<GuidName>>)

EFI_GUID gEfi<<GuidName>>Guid = EFI_<<GUID_NAME>>_GUID;

EFI_GUID_STRING (&gEfi<<GuidName>>Guid, "", "");
```

Example 23. GUID C File

3.12.3 Including a Protocol or a GUID

Any code that produces or consumes a protocol must include the protocol definitions, and any code that produces or consumes a GUID must include the GUID definitions. A protocol can be included by using the **EFI_PROTOCOL_DEFINITION ()** macro. However, it is recommended that aliases of this macro be used that advertise if the protocol is being consumed or produced. The **EFI_PROTOCOL_CONSUMER ()** macro is used by UEFI drivers to include protocol definitions that are consumed, and the **EFI_PROTOCOL_PRODUCER ()** macro is used by UEFI drivers to include protocol definitions that are produced. There is no difference between these two macros in function, but they could potentially be used by source code analysis tools in the future to examine the relationships between various UEFI drivers. A GUID is included by using

the **EFI_GUID_DEFINITION ()** macro. Example 24 shows the different macros that can be used to include the Block I/O Protocol and the UEFI global variable GUID.

```
#include EFI_PROTOCOL_DEFINITION (BlockIo)
#include EFI_PROTOCOL_PRODUCER (BlockIo)
#include EFI_PROTOCOL_CONSUMER (BlockIo)
#include EFI_GUID_DEFINITION (GlobalVariable)
```

Example 24. Including a Protocol or a GUID

3.12.4 UEFI driver Template

UEFI drivers are placed in a subdirectory below the bus they reside on in the *EDK*, **\Sample\Bus\<<BUS>>\<<DriverName>>** directory. The directory structure in this area does not have to be flat. Closely related drivers may be placed in subdirectories. The directory name for an UEFI driver is typically of the form **<<DriverName>>**. For example, the USB keyboard driver is in the **\Sample\Bus\Usb\UsbKb** subdirectory.

Simple UEFI drivers will typically have the following two files in their driver directory:

- **<<DriverName>>.h**
- **<<DriverName>>.c**

The **<<DriverName>>.h** file includes the standard UEFI include files, the UEFI driver Library declarations, and any protocol or GUID files that the driver either consumes or produces. In addition, the **<<DriverName>>.h** file contains the function prototypes of all the public APIs that are produced by the driver. The **<<DriverName>>.c** file contains the driver entry point. If an UEFI driver produces the Driver Binding Protocol, then the **<<DriverName>>.c** file typically contains the **Supported()**, **Start()**, and **Stop()** services. The **<<DriverName>>.c** file may also contain the services for the protocol(s) that the UEFI driver produces. Complex UEFI drivers that produce more than one protocol may be broken up into multiple source files. The natural organization is to place the implementation of each protocol that is produced in a separate file of the form **<<ProtocolName>>.c** or **<<DriverName>><<ProtocolName>>.c**. For example, the disk I/O driver produces the Driver Binding Protocol, the Disk I/O Protocol, and the Component Name Protocol. The **DiskIo.c** file contains the Driver Binding Protocol and Disk I/O Protocol implementations. The **ComponentName.c** file contains the implementation of the Component Name Protocol.

3.12.5 <<DriverName>>.h File

Example 25 below shows the main include file for an UEFI driver that consumes n protocols and produces m protocols. The file header comments and the function prototypes for the exported APIs have been omitted in this template. The full source to the include file for the disk I/O driver is included in Appendix D for reference. An UEFI driver include file contains the following:

- **#ifndef** / **#define** for the driver include file
- **#include** statements for the standard UEFI and UEFI driver Library include files.
- **#include** statements for all the protocols and GUIDs that are consumed by the driver.

- `#include` statements for all the protocols and GUIDs that are produced by the driver.
- `#define` for a unique signature that is used in the private context data structure (see chapter 8).
- `typedef struct` for the private context data structure (see chapter 8).
- `#define` statements to retrieve the private context data structure from each protocol that is produced (see chapter 8).
- `extern` statements for the global variables that the driver produces.
- Function prototype for the driver's entry point.
- Function prototypes for all of the APIs in the produced protocols
- `#endif` statement for the driver include file

```
#ifndef __EFI_<<DRIVER_NAME>>_H__
#define __EFI_<<DRIVER_NAME>>_H__

#include "Efi.h"
#include "EfiDriverLib.h"

//
// Driver Consumed Protocol Prototypes
//
#include EFI_PROTOCOL_CONSUMER (<<ProtocolNameC1>>)
#include EFI_PROTOCOL_CONSUMER (<<ProtocolNameC2>>)
// . . .
#include EFI_PROTOCOL_CONSUMER (<<ProtocolNameCn>>)

//
// Driver Produced Protocol Prototypes
//
#include EFI_PROTOCOL_PRODUCER (<<ProtocolNameP1>>)
#include EFI_PROTOCOL_PRODUCER (<<ProtocolNameP2>>)
// . . .
#include EFI_PROTOCOL_PRODUCER (<<ProtocolNamePm>>)

//
// Private Data Structure
//
#define <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE  EFI_SIGNATURE_32
('A','B','C','D')

typedef struct {
  UINTN                   Signature;
  EFI_HANDLE              Handle;

  //
  // Pointers to consumed protocols
  //
  EFI_<<PROTOCOL_NAME_C1>>_PROTOCOL  *<<ProtocolNameC1>>;
  EFI_<<PROTOCOL_NAME_C2>>_PROTOCOL  *<<ProtocolNameC2>>;
  // . . .
  EFI_<<PROTOCOL_NAME_Cn>>_PROTOCOL  *<<ProtocolNameCn>>;

  //
  // Produced protocols
  //
  EFI_<<PROTOCOL_NAME_P1>>_PROTOCOL  <<ProtocolNameP1>>;
  EFI_<<PROTOCOL_NAME_P2>>_PROTOCOL  <<ProtocolNameP2>>;
  // . . .
  EFI_<<PROTOCOL_NAME_Pm>>_PROTOCOL  <<ProtocolNamePm>>;

  //
  // Private functions and data fields
  //
} <<DRIVER_NAME>>_PRIVATE_DATA;

#define <<DRIVER_NAME>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_P1>>_THIS(a) \
  CR(                                                                 \
    a,                                                                \
    <<DRIVER_NAME>>_PRIVATE_DATA,                                     \
    <<ProtocolNameP1>>,                                               \
    <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE                            \
  )

#define <<DRIVER_NAME>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_P2>>_THIS(a) \
  CR(                                                                 \
    a,                                                                \
    <<DRIVER_NAME>>_PRIVATE_DATA,                                     \
    <<ProtocolNameP2>>,                                               \
    <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE                            \
  )
```

```
// . . .

#define <<DRIVER_NAME>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Pm>>_THIS(a) \
  CR(                                                                 \
    a,                                                                \
    <<DRIVER_NAME>>_PRIVATE_DATA,                                     \
    <<ProtocolNamePm>>,                                               \
    <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE                            \
  )

//
// Required Global Variables
//
extern EFI_DRIVER_BINDING_PROTOCOL       g<<DriverName>>DriverBinding;

//
// Optional Global Variables
//
extern EFI_COMPONENT_NAME_PROTOCOL       g<DriverName>>ComponentName;
extern EFI_DRIVER_CONFIGURATION_PROTOCOL
g<DriverName>>DriverConfiguration;
extern EFI_DRIVER_DIAGNOSTICS_PROTOCOL   g<DriverName>>DriverDiagnostics;

//
// Function prototype for the driver's entry point
//
EFI_STATUS
EFIAPI
<<DriverName>>DriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  );

//
// Function ptototypes for the APIs in the Produced Protocols
//

#endif
```

Example 25. Driver Include File Template

3.12.6 <<DriverName>>.c File

The template in Example 26 below shows the main source file for an UEFI driver. The file header comments and function header comments have been omitted in this template. This template contains empty protocol functions. The functions from the various protocols that an UEFI driver may produce are discussed in later chapters. The full source to the include file for the disk I/O driver is included in Appendix D for reference. A UEFI source file contains the following:

- **#include** statement for **<<DriverName>>.h**.
- Global variable declarations
- Declaration of the UEFI driver's entry point function
- The UEFI driver entry point function
- The **Supported()**, **Start()**, and **Stop()** functions
- Implementation of the APIs from the produced protocols

```
#include "<<DriverName>>.h"

EFI_DRIVER_BINDING_PROTOCOL g<<DriverName>>DriverBinding = {
  <<DriverName>>DriverBindingSupported,
  <<DriverName>>DriverBindingStart,
  <<DriverName>>DriverBindingStop,
  <<DriverVersion>>,
  NULL,
  NULL
};

EFI_DRIVER_ENTRY_POINT (<<DriverName>>DriverEntryPoint)

EFI_STATUS
EFIAPI
<<DriverName>>DriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  return EfiLibInstallAllDriverProtocols (
           ImageHandle,
           SystemTable,
           &g<<DriverName>>DriverBinding,
           ImageHandle,
           &g<<DriverName>>ComponentName,
           &g<<DriverName>>DriverConfiguration,
           &g<<DriverName>>DriverDiagnostics
           );
}

EFI_STATUS
EFIAPI
<<DriverName>>DriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )
{
}

EFI_STATUS
EFIAPI
<<DriverName>>DriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )
{
}

EFI_STATUS
EFIAPI
<<DriverName>>DriverBindingStop (
  IN  EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN  EFI_HANDLE                   ControllerHandle,
  IN  UINTN                        NumberOfChildren,
  IN  EFI_HANDLE                   *ChildHandleBuffer
  )
{
}

//
// Implementations of the APIs in the produced protocols
// The following template is for the mth function of the nth protocol
produced
```

```
// It also shows how to retrieve the private context structure from this
arg
//
EFI_STATUS
EFIAPI
<<DriverName>><<ProtocolNamePn>><<FunctionNameM>> (
  IN EFI_<<PROTOCOL_NAME_PN>>_PROTOCOL *This,
  //
  // Additional function arguments here.
  //
  )
{
  <<DRIVER_NAME>>_PRIVATE_DATA  *Private;

  //
  // Use This pointer to retrieve the private context structure
  //
  Private = <<DRIVER_NAME>>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Pn>>_THIS
(This);
}
```

Example 26. Driver Implementation Template

3.12.7 <<ProtocolName>>.c or <<DriverName>><<ProtocolName>>.c File

More complex UEFI drivers may break the implementation into several source files. The natural boundary is to implement one protocol per file. The template in Example 27 below shows the main source file for one protocol of an UEFI driver. The file header comments and function header comments have been omitted in this template. This template shows only empty protocol functions. The functions from the various protocols that an UEFI driver may produce are discussed in later chapters. The full source to the Component Name Protocol for the disk I/O driver is included in Appendix D for reference. An UEFI protocol source file contains the following:

- **`#include`** statement for **`<<DriverName>>.h`**.
- Global variable declaration. This declaration applies only to protocols such as the Component Name, Driver Configuration, and Driver Diagnostics Protocols. Protocols that produce I/O services should never be declared as a global variable. Instead, they are declared in the private context structure that is dynamically allocated in the **`Start()`** function (see chapter 8).
- Implementation of the APIs from the produced protocols.

```
#include "<<DriverName>>.h"

//
// Protocol Global Variables
//
EFI_<<PROTOCOL_NAME_PN>>_PROTOCOL g<<DriverName>><<ProtocolNamePn>> = {
  // . . .
};

//
// Implementations of the APIs in the produced protocols
// The following template is for the mth function of the nth protocol
produced
// It also shows how to retrieve the private context structure from the
This arg
//
EFI_STATUS
EFIAPI
<<DriverName>><<ProtocolNamePn>><<FunctionName1M>> (
  IN EFI_<<PROTOCOL_NAME_PN>>_PROTOCOL *This,
  //
  // Additional function arguments here.
  //
  )
{
  <<DRIVER_NAME>>_PRIVATE_DATA  Private;

  //
  // Use This pointer to retrieve the private context structure
  //
  Private = <<DRIVER_NAME>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Pn>>_THIS
(This);
}
```

Example 27. Protocol Implementation Template

3.12.8 UEFI driver Library

Most UEFI drivers use the UEFI driver Library because it simplifies the implementation and reduces the size of both the source code and the executable. All UEFI drivers that use the UEFI driver Library must include the **`EfiDriverLib.h`** file. There are three different library initialization functions that are available from the UEFI driver Library. All three of these functions initialize the global variables **`gST`**, **`gBS`**, and **`gRT`**. These three global variables are pointers to the EFI System Table, the EFI Boot Services Table, and the EFI Runtime Services Table, respectively. There are also a number of function and macros available from the UEFI driver Library. They are all documented in the *UEFI driver Library Specification*. The code fragment in Example 28 below shows examples of the different UEFI driver Library initialization functions. The first example is typically used by services drivers that do not follow the UEFI Driver Model. The second example is typically used by simple UEFI drivers that follow the UEFI Driver Model and do not produce the optional driver-related protocols. The last example is typically used by more complex UEFI drivers that follow the UEFI Driver Model and produce at least one of the optional driver-related protocols.

```
//
// Initialize a service driver
//
EfiInitializeDriverLib (ImageHandle, SystemTable);

//
// Initialize a simple UEFI driver that follows the UEFI Driver Model
//
EfiLibInstallDriverBinding (
  ImageHandle,
  SystemTable,
  gDiskIoDriverBinding,
  ImageHandle
  );

//
// Initialize a complex UEFI driver that follows the UEFI Driver Model
//
EfiLibInstallAllDriverProtocols (
  ImageHandle,
  SystemTable,
  gDiskIoDriverBinding,
  ImageHandle,
  gDiskIoComponentName,
  gDiskIoDriverConfiguration,
  gDiskIoDriverDiagnostics
  );
```

Example 28. Initializing the UEFI driver Library

4 EFI Services

An UEFI driver has a number of EFI Boot Services and EFI Runtime Services that are available to perform its functions. These services can be grouped into the following three areas:

- Commonly used services
- Rarely used services
- Services that should not be used from an UEFI driver

Table 55, Table 56, and Table 57 in Appendix C list each of these categories. The full function prototypes and descriptions for each of the services and their arguments described in this chapter are available in chapters 5 and 6 of the *UEFI 2.0 Specification*.

4.1 Services That UEFI drivers Commonly Use

Table 16 contains the list of UEFI services that are commonly used by UEFI drivers. The following sections provide a brief description of each service along with code examples on how they are typically used by UEFI drivers. This table is also present in Appendix C for reference purposes.

Table 16. UEFI Services That Are Commonly Used by UEFI drivers

Type	Service	Type	Service
BS	gBS->AllocatePool()	BS	gBS->InstallMultipleProtocolInterfaces()
BS	gBS->FreePool()	BS	gBS->UninstallMultipleProtocolInterfaces()
BS	gBS->AllocatePages()	BS	gBS->LocateHandleBuffer()
BS	gBS->FreePages()	BS	gBS->LocateProtocol()
BS	gBS->SetMem()	BS	gBS->OpenProtocol()
BS	gBS->CopyMem()	BS	gBS->CloseProtocol()
BS	gBS->CreateEvent()	BS	gBS->OpenProtocolInformation()
BS	gBS->CreateEventEx()		
BS	gBS->CloseEvent()	BS	gBS->RaiseTPL()
BS	gBS->SignalEvent()	BS	gBS->RestoreTPL()
BS	gBS->SetTimer()	BS	gBS->Stall()
BS	gBS->CheckEvent()		

4.1.1 Memory Services

4.1.1.1 gBS->AllocatePool() and gBS->FreePool()

These services are used by UEFI drivers to allocate and free small buffers that are guaranteed to be aligned on an 8-byte boundary. These services are ideal for allocating and freeing data structures. Because the allocated buffers are guaranteed to be on an 8-byte boundary, an alignment fault will not be generated on Itanium-based platforms as long as all the fields of the data structure are natural aligned (not packed).

When a buffer is allocated on an IA-32 Instruction Set Architecture platform, the buffer will be allocated below 4 GB, so it is guaranteed to be accessible by an IA-32 instruction set architecture that is executing in flat physical mode. When a buffer is allocated on an Itanium-based platform, the buffer will be allocated somewhere in the 64-bit address space that is available to the Itanium processor in physical mode. This location means that buffers above 4 GB may be allocated on Itanium-based platforms if there is system memory above 4 GB. It is important to note that care must be taken when pointers are converted on Itanium-based platforms. All UEFI drivers must be aware that pointers may contain values above 4 GB, and care must be taken never to strip the upper address bits. If the upper address bits are stripped, then the driver will work on IA-32 systems and Itanium-based platforms with small memory configurations, but not on Itanium-based platforms with larger memory configurations.

EFI Boot Service drivers will typically allocate and free buffers of type **`EfiBootServicesData`**, and UEFI runtime drivers will typically allocate and free buffers of type **`EfiRuntimeServicesData`**. Most drivers that follow the UEFI Driver Model will allocate private context structures in their **`Start()`** function and will free them in their **`Stop()`** function. UEFI drivers may also dynamically allocate and free buffers as different I/O operations are performed. To prevent memory leaks, every allocation operation must have a corresponding free operation. The code fragment in Example 29 shows how these services can be used to allocate and free a buffer for a data structure from **`EfiBootServicesData`** memory. In addition, it shows how the same allocations can be performed using services from the UEFI driver Library. The UEFI driver Library provides services that reduce the size of the driver and make the driver code easier to read and maintain.

```
EFI_STATUS      Status;
IDE_BLK_IO_DEV  *IdeBlkIoDevice;

//
// Allocate a buffer for a data structure
//
Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (IDE_BLK_IO_DEV),
                (VOID**)&IdeBlkIoDevice
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Allocate the same buffer using an EFI Library Function
//
IdeBlkIoDevice = EfiLibAllocatePool (sizeof (IDE_BLK_IO_DEV));
if (IdeBlkIoDevice == NULL) {
  return EFI_OUT_OF_RESOURCES;
}

//
// Allocate the same buffer using an EFI Library Function that also
// initializes the contents of the buffer with zeros
//
IdeBlkIoDevice = EfiLibAllocateZeroPool (sizeof (IDE_BLK_IO_DEV));
if (IdeBlkIoDevice == NULL) {
  return EFI_OUT_OF_RESOURCES;
}

//
// Free the allocated buffer
//
Status = gBS->FreePool (IdeBlkIoDevice);
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 29. Allocate and Free Pool

4.1.1.2 gBS->AllocatePages() and gBS->FreePages()

These services are used by UEFI drivers to allocate and free larger buffers that are guaranteed to be aligned on a 4 KB boundary. These services allow buffers to be allocated at any available address, at specific addresses, or below a specific address. However, UEFI drivers should not make any assumptions about the organization of system memory, so allocating from specific addresses or below specific addresses is strongly discouraged. As a result, buffers are typically allocated with an allocation type of **AllocateAnyPages**, and allocation types of **AllocateMaxAddress** and **AllocateAddress** are not used.

When an allocation type of **AllocateAnyPages** is used on IA-32 platforms, the buffer will be allocated below 4 GB, so it is guaranteed to be accessible by an IA-32 processor that is executing in flat physical mode. When an allocation type of **AllocateAnyPages** is used on Itanium-based platforms, the buffer will be allocated somewhere in the 64-bit address space that is available to the Itanium processor in physical mode. This location means that buffers above 4 GB may be allocated on Itanium-based platforms if there is system memory above 4 GB. It is important to note this possible allocation because care must be taken when the physical address is converted to a pointer on

Itanium-based platforms. All UEFI drivers must be aware that pointers may contain values above 4 GB, and care must be taken never to strip the upper address bits. If the upper address bits are stripped, then the driver will work on IA-32 systems and Itanium-based platforms with small memory configurations, but not on Itanium-based platforms with larger memory configurations.

EFI Boot Service drivers will typically allocate and free buffers of type **EfiBootServicesData**, and UEFI runtime drivers will typically allocate and free buffers of type **EfiRuntimeServicesData**. To prevent memory leaks, every allocation operation must have a corresponding free operation. The code fragment in Example 30 shows how these services can be used to allocate and free a buffer for a data structure from **EfiBootServicesData** memory.

```
EFI_STATUS            Status;
EFI_PHYSICAL_ADDRESS  PhysicalBuffer;
UINTN                 Size;
VOID                  *Buffer;

//
// Allocate the number of pages to hold Size bytes and return in
PhysicalBuffer
//
Status = gBS->AllocatePages(
               AllocateAnyPages,
               EfiBootServicesData,
               EFI_SIZE_TO_PAGES(Size),
               &PhysicalBuffer
               );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Convert the physical address to a pointer. This mechanism is the best
one
// that works on all IA-32 and Itanium-based platforms.
//
Buffer = (VOID *)(UINTN)PhysicalBuffer;

//
// Free the allocated buffer
//
Status = gBS->FreePages (RomBuffer, EFI_SIZE_TO_PAGES(RomBarSize));
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 30. Allocate and Free Pages

4.1.1.3 gBS->SetMem()

This service is used to initialize the contents of a buffer with a specified value. UEFI drivers most commonly use this service to zero an allocated buffer, but it can be used to fill a buffer with other values too. The code fragment in Example 31 shows the same example from Example 29, but it uses this service to zero the contents of the allocated buffer.

```
EFI_STATUS      Status;
IDE_BLK_IO_DEV  *IdeBlkIoDevice;

//
// Allocate a buffer for a data structure
//
Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (IDE_BLK_IO_DEV),
                (VOID**)&IdeBlkIoDevice
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Zero the contents of the allocated buffer
//
gBS->SetMem (IdeBlkIoDevice, sizeof (IDE_BLK_IO_DEV), 0);
```

Example 31. Allocate and Free Buffer

4.1.1.4 gBS->CopyMem()

This service copies a buffer from one location to another. This service handles both aligned and unaligned buffers, and it even handles the rare case when buffers are overlapping. In the overlapping case, the requirement is that the destination buffer on exit from this service matches the contents of the source buffer on entry to this service. The code fragment in Example 32 shows how this service is typically used.

```
EFI_STATUS    Status;
MY_STRUCTURE  *SourceStructure;
MY_STRUCTURE  *DestinationStructure;

//
// Allocate a buffer for the destination buffer
//
DestinationStructure = EfiLibAllocatePool (sizeof (MY_STRUCTURE));
if (DestinationStructure == NULL) {
  return EFI_OUT_OF_RESOURCES;
}

//
// Copy the source buffer to the destination buffer
//
gBS->CopyMem (DestinationBuffer, SourceBuffer, sizeof (MY_STRUCTURE));
```

Example 32. Allocate and Copy Buffer

4.1.2 Handle Database and Protocol Services

4.1.2.1 gBS->InstallMultipleProtocolInterfaces() and gBS->UninstallMultipleProtocolInterfaces()

These services are used to do the following:

- Add handles to the handle database.

- Remove handles from the handle database.
- Add protocols to an existing handle in the handle database.
- Remove protocols from an existing handle in the handle database.

A handle in the handle database contains one or more protocols. A handle is not allowed to have zero protocols, and a handle is not allowed to have more than one instance of the same protocol on the same handle. A handle is added to the handle database when the first protocol is added. A handle is removed from the handle database when the last protocol is removed from the handle. These services support adding and removing more than one protocol at a time. If one protocol fails to be added to a handle, then none of the protocols are added to the handle. If one protocol fails to be removed from a handle, then none of the protocols are removed from the handle. The protocols are represented by a pointer to a protocol GUID and a pointer to the protocol interface. These services will parse pairs of arguments until a **NULL** pointer for the protocol GUID parameter is encountered. It is recommended that these services be used instead of **gBS->InstallProtocolInterface()** and **gBS->UninstallProtocolInterface()** because it results in small executables with source code that is easier to read. In addition, the **gBS->InstallMultipleProtocolInterfaces()** service will check to see if the same device path is being installed onto more than one handle in the handle database. This operation is not legal, so this additional error checking was added to section 6.3.1 of the *UEFI 2.0 Specification*.

UEFI drivers are required to add the **EFI_DRIVER_BINDING_PROTOCOL** to the handle database in their driver entry point. They may also add the following in the driver entry point:

- **EFI_DRIVER_CONFIGURATION_PROTOCOL**
- **EFI_DRIVER_DIAGNOSTICS_PROTOCOL**
- **EFI_COMPONENT_NAME_PROTOCOL**

These protocols are optional for some platforms and required for others. For example, *DIG64 requires* these protocols for Itanium-based platforms. If an UEFI driver is unloadable, then the protocols that were added in the driver entry point must be removed in the driver's **Unload()** function. The code fragment in Example 33 shows how these services would be used in a driver entry point and **Unload()** function.

```
EFI_DRIVER_BINDING_PROTOCOL  gMyDriverBinding = {
  MySupported,
  MyStart,
  MyStop,
  0x10,
  NULL,
  NULL
};

EFI_COMPONENT_NAME_PROTOCOL  gMyComponentName = {
  MyGetDriverName,
  MyGetControllerName,
  "eng"
};

EFI_HANDLE  ImageHandle;

//
// Install the Driver Binding Protocol and the Component Name Protocol
// onto the image handle that is passed into the driver entry point
//
Status = gBS->InstallMultipleProtocolInterfaces (
                &ImageHandle,
                &gEfiDriverBindingProtocolGuid, &gMyDriverBinding,
                &gEfiComponentNameProtocolGuid, &gMyComponentName,
                NULL
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Uninstall the Driver Binding Protocol and the Component Name Protocol
// from the handle that is passed into the Unload() function.
//
Status = gBS->UninstallMultipleProtocolInterfaces (
                ImageHandle,
                &gEfiDriverBindingProtocolGuid, &gMyDriverBinding,
                &gEfiComponentNameProtocolGuid, &gMyComponentName,
                NULL
                );
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 33. Install Driver Protocols

EFI device drivers will add protocols for I/O services to existing handles in the handle database in their **`Start()`** function and will remove those same protocols from those same handles in their **`Stop()`** function. UEFI bus drivers may add protocols to existing handles, but they are also responsible for creating handles for the child device on that bus. This responsibility means that the UEFI bus driver will typically add the **`EFI_DEVICE_PATH_PROTOCOL`** and an I/O abstraction for the bus type that the bus driver manages. For example, the PCI bus driver will create child handles with both the **`EFI_DEVICE_PATH_PROTOCOL`** and the **`EFI_PCI_IO_PROTOCOL`**. The bus driver may also optionally add the **`EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL`** to the child handles. The code fragment in Example 34 shows an example of a how a child handle can be created and additional protocols added and then destroyed.

```
EFI_HANDLE  ChildHandle;

//
// Add Device Path Protocol and Block I/O Protocol to a new handle
//
ChildHandle = NULL;
Status = gBS->InstallMultipleProtocolInterfaces (
                &ChildHandle,
                &gEfiDevicePathProtocol, DevicePath,
                &gEfiBlockIoProtocol,    BlockIo
                NULL
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Add the Disk I/O Protocol to the new handle created in the previous
call
//
Status = gBS->InstallMultipleProtocolInterfaces (
                &ChildHandle,
                &gEfiDiskIoProtocol, DiskIo
                NULL
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Remove Device Path Protocol, Block I/O Protocol, and Disk I/O Protocol
// from the handle created above. Because this call will remove all the
// protocols from the handle, the handle will be removed from the handle
// database
//
Status = gBS->UnistallMultipleProtocolInterfaces (
                ChildHandle,
                &gEfiDevicePathProtocolGuid, DevicePath,
                &gEfiBlockIoProtocolGuid,    BlockIo,
                &gEfiDiskIoProtocolGuid,     DiskIo
                NULL
                );
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 34. Install I/O Protocols

Some UEFI device drivers will add protocols to handles in the handle database with a **NULL** protocol interface. This case is known as a tag GUID because there are no data fields or services associated with the GUID. The code fragment in Example 35 shows an example of a how a tag GUID for hot-plug devices can be added and removed from a controller handle in the handle database.

```
EFI_HANDLE  ControllerHandle;

//
// Add the hot-plug device GUID to ControllerHandle
//
Status = gBS->InstallMultipleProtocolInterfaces (
                &ControllerHandle,
                &gEfiHotPlugDeviceGuid, NULL,
                NULL
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Remove the hot-plug device GUID from ControllerHandle
//
Status = gBS->UninstallMultipleProtocolInterfaces (
                ControllerHandle,
                &gEfiHotPlugDeviceGuid, NULL,
                NULL
                );
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 35. Install Tag GUID

Note: *When an attempt is made to remove a protocol interface from a handle in the handle database, the UEFI core firmware will check to see if any other UEFI drivers are currently using the services of the protocol that is about to be removed. If UEFI drivers are using that protocol interface, then the UEFI core firmware will attempt to stop those UEFI drivers with a call to* ***gBS->DisconnectController()****. If the call to* ***gBS->DisconnectController()*** *fails, then the UEFI core firmware will have to call* ***gBS->ConnectController()*** *to put the handle database back into the same state that it was in prior to the original call to* ***gBS->UninstallMultipleProtocolInterfaces()****. This call to* ***gBS->ConnectController()*** *has the potential to cause issues upon re-entry in UEFI drivers that must be handled in the UEFI driver. Please see chapter 24 for recommendations on how to test UEFI drivers.*

4.1.2.2 gBS->LocateHandleBuffer()

This service retrieves a list of handles that meet a search criterion from the handle database. The following are the search options:

- Retrieve all the handles in the handle database.
- Retrieve the handles that support a specific protocol.
- Retrieve the handles that are in the notified state based on a prior **gBS->RegisterProtocolNotify()**.

It is not recommended that UEFI drivers use **gBS->RegisterProtocolNotify()**, so only the first two cases will be covered here. The buffer that is returned by this service is allocated by the service, so an UEFI driver that uses this service is responsible for freeing the returned buffer when the UEFI driver no longer requires its contents. This service, along with **gBS->ProtocolsPerHandle()** and **gBS->OpenProtocolInformation()**, can be used to traverse the contents of the entire

handle database. The algorithm for performing this traversal is in section 6.3.1 of the *UEFI 2.0 Specification*.

The code fragment in Example 36 shows how all the handles in the handle database can be retrieved.

```
EFI_STATUS  Status;
UINTN       HandleCount;
EFI_HANDLE  *HandleBuffer;

//
// Retrieve the list of all the handles in the handle database. The
number
// of handles in the handle database is returned in HandleCount, and the
// array of handle values is returned in HandleBuffer.
//
Status = gBS->LocateHandleBuffer (
                AllHandles,
                NULL,
                NULL,
                &HandleCount,
                &HandleBuffer
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Free the array of handles that was allocated by
gBS->LocateHandleBuffer()
//
gBS->FreePool (HandleBuffer);
```

Example 36. Locate All Handles

The code fragment in Example 37 shows how all the handles that support the Block I/O Protocol can be retrieved and how the individual Block I/O Protocol instances can be retrieved using **`gBS->OpenProtrocol()`**.

```
EFI_STATUS              Status;
UINTN                   HandleCount;
EFI_HANDLE              *HandleBuffer;
UINTN                   Index;
EFI_BLOCK_IO_PROTOCOL   *BlockIo;

//
// Retrieve the list of handles that support the Block I/O Protocol from
// the handle database. The number of handles that support the Block I/O
// Protocol is returned in HandleCount, and the array of handle values is
// returned in HandleBuffer.
//
Status = gBS->LocateHandleBuffer (
                ByProtocol,
                &gEfiBlockIoProtocolGuid,
                NULL,
                &HandleCount,
                &HandleBuffer
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Loop through all the handles that support the Block I/O Protocol, and
// retrieve the instance of the Block I/O Protocol.
//
for (Index = 0; Index < HandleCount; Index++) {
  Status = gBS->OpenProtocol (
                  HandleBuffer[Index],
                  &gEfiBlockIoProtocolGuid,
                  (VOID **)&BlockIo,
                  ImageHandle,
                  NULL,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  //
  // BlockIo can be used here to make block I/O service requests.
  //

}
//
// Free the array of handles that was allocated by
gBS->LocateHandleBuffer()
//
gBS->FreePool (HandleBuffer);
```

Example 37. Locate Block I/O Protocol Handles

4.1.2.3 gBS->LocateProtocol()

This service finds the first instance of a protocol interface in the handle database. This service is typically used by UEFI drivers to retrieve service protocols on service handles that are guaranteed to have at most one instance of the protocol in the handle database. The list of service protocols that are defined in the *UEFI 2.0 Specification* includes the following:

- **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL**
- **EFI_UNICODE_COLLATION_PROTOCOL**

- **EFI_BIS_PROTOCOL**
- **EFI_DEBUG_SUPPORT_PROTOCOL**
- **EFI_DECOMPRESS_PROTOCOL**
- **EFI_EBC_PROTOCOL**

If there is a chance that more than one instance of a protocol may exist in the handle database, then **gBS->LocateHandleBuffer()** should be used. This service also supports retrieving protocols that have been notified with **gBS->RegisterProtocolNotify()**. It is not recommended that UEFI drivers use **gBS->RegisterProtocolNotify()**, so this case will not be covered here. The code fragment in Example 38 shows how this service can be used to retrieve the **EFI_DECOMPRESS_PROTOCOL**.

```
EFI_STATUS                Status;
EFI_DECOMPRESS_PROTOCOL   *Decompress;

Status = gBS->LocateProtocol(
                &gEfiDecompressProtocolGuid,
                NULL,
                (VOID **)&Decompress
                );
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 38. Locate Decompress Protocol

4.1.2.4 gBS->OpenProtocol() and gBS->CloseProtocol()

These two services are used by UEFI drivers to acquire and release the protocol interfaces that the UEFI drivers require to produce their services. These services are some of the more complex services in EFI, but using them correctly is required for UEFI drivers to produce their I/O abstractions and work well with other UEFI drivers. UEFI applications and UEFI OS loaders can also use these services, but the discussion here will concentrate on the different ways that UEFI drivers use these services.

gBS->OpenProtocol() is typically used by the **Supported()** and **Start()** functions of an UEFI driver to retrieve a protocol interface that is installed on a handle in the handle database. A brief description of the parameters and return codes is presented here. A more complete description can be found in section 6.3.1 of the *UEFI 2.0 Specification*. The function prototype of this service is shown below for reference.

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

The only **OUT** parameter is the pointer to the protocol interface. All the other parameters are **IN** parameters that tell the UEFI core why the protocol interface is being retrieved and by whom. The UEFI core uses these **IN** parameters to track how each protocol interface is being used. This tracking information can be retrieved by using the **gBS->OpenProtocolInformation()** service. *AgentHandle* and *ControllerHandle* describe who is opening the protocol interface, and the *Attributes* parameter tells why the protocol interface is being opened. For UEFI drivers, the *AgentHandle* parameter is typically the *DriverBindingHandle* field from the **EFI_DRIVER_BINDING_PROTOCOL**. Also, *Handle* and *ControllerHandle* are typically the same handle. The only exception is when a bus driver is opening a protocol on behalf of a child controller. The attributes that are used by UEFI drivers are listed below. It is very important that UEFI drivers use the correct attributes when a protocol interface is opened.

TEST_PROTOCOL Tests to see if a protocol interface is present on a handle. Typically used in the **Supported()** service of an UEFI driver.

GET_PROTOCOL Retrieves a protocol interface from a handle.

BY_DRIVER Retrieves a protocol interface from a handle and marks that interface so it cannot be opened by other UEFI drivers or UEFI applications unless the other UEFI driver agrees to release the protocol interface. This attribute is the most commonly used.

BY_DRIVER | EXCLUSIVE
Retrieves a protocol interface from a handle and marks the interface so it cannot be opened by other UEFI drivers or UEFI applications. This protocol will not be released until the driver that opened this attribute chooses to close it. This attribute is used very rarely.

BY_CHILD_CONTROLLER
Only used by bus drivers. A bus driver is required to open the parent I/O abstraction on behalf of each child controller that the bus driver produces. This requirement allows the UEFI core to keep track of the parent/child relationships no matter how complex the bus hierarchies become.

The status code returned is also very important and must be examined by UEFI drivers that use this service. The typical return codes are listed below.

EFI_SUCCESS The protocol interface was retrieved.

EFI_UNSUPPORTED The protocol interface is not present on the handle.

EFI_ALREADY_STARTED
The UEFI driver has already retrieved the protocol interface.

EFI_ACCESS_DENIED A different UEFI driver has already retrieved the protocol interface.

EFI_INVALID_PARAMETER
One of the parameters is invalid.

The **gBS->CloseProtocol()** service removes an element from the list of agents that are consuming a protocol interface. UEFI drivers are required to close each protocol that they open, which is typically done in the **Stop()** function.

The code fragment in Example 39 shows the most common use of these services. This example shows a stripped-down version of a **Support()** function for a PCI device driver. It opens the PCI I/O Protocol **BY_DRIVER**, and then closes it if the open operation worked. If this driver wanted to open the PCI I/O Protocol exclusively, then an attribute of **BY_DRIVER | EXCLUSIVE** should be used. There are only a very few instances where **BY_DRIVER | EXCLUSIVE** should be used. These are cases where an UEFI driver actually wants to gain exclusive access to a protocol, even if it requires stopping other UEFI drivers to do so. This operation can be dangerous if the system requires the services produced by the UEFI drivers that are stopped. One example is the debug port driver that opens the Serial I/O Protocol with the **BY_DRIVER | EXCLUSIVE** attribute. This attribute allows a debugger to take control of a serial port even if it is being used as a console device. If this device is the only console device in the system, then the user will lose the only console device when the debug port driver is started.

```
EFI_STATUS
XyzDriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath
  )

{
  EFI_STATUS           Status;
  EFI_PCI_IO_PROTOCOL  *PciIo;

  Status = gBS->OpenProtocol (
                 ControllerHandle,
                 &gEfiPciIoProtocolGuid,
                 (VOID **)&PciIo,
                 This->DriverBindingHandle,
                 ControllerHandle,
                 EFI_OPEN_PROTOCOL_BY_DRIVER
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  gBS->CloseProtocol (
         ControllerHandle,
         &gEfiPciIoProtocolGuid,
         This->DriverBindingHandle,
         ControllerHandle
         );

  return Status;
}
```

Example 39. Open Protocol BY_DRIVER

The code fragment in Example 40 shows the same example as above, but it tests only for the presence of the PCI I/O Protocol using the **TEST_PROTOCOL** attribute. When **TEST_PROTOCOL** is used, the protocol does not have to be closed because a protocol interface is not returned when this open mode is used.

```
EFI_STATUS
XyzDriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath
  )

{
  EFI_STATUS           Status;

  Status = gBS->OpenProtocol (
                 ControllerHandle,
                 &gEfiPciIoProtocolGuid,
                 NULL,
                 This->DriverBindingHandle,
                 ControllerHandle,
                 EFI_OPEN_PROTOCOL_TEST_PROTOCOL
                );
  return Status;
}
```

Example 40. Open Protocol by TEST_PROTOCOL

The code fragment in Example 41 shows the same example as above, but it retrieves the PCI I/O Protocol using the **GET_PROTOCOL** attribute. When **GET_PROTOCOL** is used, the protocol does not have to be closed.

Note: *It can be dangerous to use this open mode because a protocol may be removed at any time, and a driver that uses **GET_PROTOCOL** may attempt to use a stale protocol interface. There are a few places where it has to be used. An UEFI driver should be designed to use **BY_DRIVER** as its first choice. However, there are cases where a different UEFI driver has already opened the protocol that is required **BY_DRIVER**. The next best choice is to use **GET_PROTOCOL**. This scenario may occur when protocols are layered on top of each other, so that each layer uses the services of the layer immediately below. Each layer immediately below is opened **BY_DRIVER**. If a layer ever needs to skip around a layer to a lower-level service, then it is safe to use **GET_PROTOCOL** because the driver will be informed through the layers if the lower-level protocol is removed.*

The best example of this case in the *EDK* is the FAT driver. The FAT driver uses the services of the Disk I/O Protocol to access the contents of the disk. However, the Disk I/O Protocol does not have a flush service. Only the Block I/O Protocol has a flush service. The disk I/O driver opens the Block I/O Protocol **BY_DRIVER**, so the FAT driver is also not allowed to open the Block I/O Protocol **BY_DRIVER**. Instead, the FAT driver must use **GET_PROTOCOL**. This method is safe because the FAT driver will be indirectly notified if the Block I/O Protocol is removed when the Disk I/O Protocol is removed in response to the Block I/O Protocol being removed.

```
EFI_STATUS
XyzDriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath
  )

{
  EFI_STATUS           Status;
  EFI_PCI_IO_PROTOCOL  *PciIo;

  Status = gBS->OpenProtocol (
                 ControllerHandle,
                 &gEfiPciIoProtocolGuid,
                 (VOID **)&PciIo,
                 This->DriverBindingHandle,
                 ControllerHandle,
                 EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Use the services of the PCI I/O Protocol here
  //
}
```

Example 41. Open Protocol by GET_PROTOCOL

The code fragment in Example 42 shows an example of a PCI bus driver opening the PCI Root Bridge I/O Protocol on behalf of a child PCI controller. This example shows the **BY_CHILD_CONROLLER** attribute being used. This attribute is typically used in the **Start()** function after the child handle has been created using **gBS->InstallMultipleProtocolInterfaces()**.

```
EFI_DRIVER_BINDING_PROTOCOL      gPciBusDriverBinding;

EFI_STATUS                       Status;
EFI_HANDLE                       ChildHandle;
EFI_DEVICE_PATH_PROTOCOL         *DevicePath;
EFI_PCI_IO_PROTOCOL              *PciIo;
EFI_HANDLE                       ControllerHandle;
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL  *PciRootBridgeIo;

ChildHandle = NULL;
Status = gBS->InstallMultipleProtocolInterfaces (
                &ChildHandle,
                &gEfiDevicePathProtocolGuid,  DevicePath,
                &gEfiPciIoProtocolGuid,       PciIo,
                NULL
                );
if (EFI_ERROR (Status)) {
  return Status;
}

Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiPciRootBridgeIoProtocolGuid,
                (VOID **)&PciRootBridgeIo,
                gPciBusDriverBinding.DriverBindingHandle,
                ChildHandle,
                EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER
                );
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 42. Open Protocol BY_CHILD_CONTROLLER

4.1.2.5 gBS->OpenProtocolInformation()

The handle database contains the following:

- List of handles
- List of protocols on each handle
- List of agents that are currently using each protocol

This service retrieves the list of agents that are currently using a specific protocol interface that is registered in the handle database. An agent is an UEFI driver or an UEFI application that is using the services of a protocol interface. The **gBS->OpenProtocol()** service adds agents to the list, and the **gBS->CloseProtocol()** service removes agents from the list. An UEFI driver will typically use this service to find the list of child handles that the UEFI driver may have produced in previous calls to the **Start()**. The return buffer is allocated by the service, so the caller must free the buffer when the caller does not need the return buffer anymore. The code fragment in Example 43 retrieves the list of agents that are using the Serial I/O Protocol on a controller handle. This step is followed by a loop that counts the number of children that have been produced from the Serial I/O Protocol. The final step is to free the return buffer. This algorithm can be used by bus drivers that produce at most one child handle, and they need to check to see if a child handle has already been produced.

```
EFI_STATUS                              Status;
EFI_HANDLE                              ControllerHandle;
EFI_OPEN_PROTOCOL_INFORMATION_ENTRY     *OpenInfo;
UINTN                                   EntryCount;
UINTN                                   Index;
UINTN                                   NumberOfChildren;

Status = gBS->OpenProtocolInformation (
                ControllerHandle,
                &gEfiSerialIoProtocolGuid,
                &OpenInfo,
                &EntryCount
                );
if (EFI_ERROR (Status)) {
  return Status;
}

for (Index = 0, NumberOfChildren = 0; Index < EntryCount; Index++) {
  if (OpenInfo[Index].Attributes & EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER)
{
    NumberOfChildren++;
  }
}

gBS->FreePool (OpenInfoBuffer);
```

Example 43. Open Protocol Information

4.1.3 Task Priority Level Services

4.1.3.1 gBS->RaiseTPL() and gBS->RestoreTPL()

These two services are used to raise and restore the Task Priority Level (TPL) of the system. The most common use of these services is to implement a simple lock, or critical section, on global data structures. UEFI firmware, applications, and drivers all run on one thread on one processor. However, UEFI firmware does support a single timer interrupt. Because UEFI code can run in interrupt context, it is possible that a global data structure can be accessed from both normal context and interrupt context. As a result, global data structures must be protected by a lock. The code fragment in Example 44 shows how these services can be used to implement a lock when the contents of a global variable are modified. The timer interrupt is blocked at **EFI_TPL_HIGH_LEVEL**, so most locks raise to this level.

```
UINT32  gCounter;

EFI_TPL  OldTpl;

//
// Raise the Task Priority Level to EFI_TPL_HIGH_LEVEL to block timer
// interrupts
//
OldTpl = gBS -> RaiseTPL(EFI_TPL_HIGH_LEVEL);

//
// Increment the global variable now that it is safe to do so.
//
gCounter++

//
// Restore the Task Priority Level to its original level
//
gBS -> RestoreTPL(OldTpl);
```

Example 44. Global Lock

These services are also used to raise the TPL during a blocking I/O transaction. Most drivers are required to raise the TPL to **EFI_TPL_NOTIFY** during blocking I/O operations. See Example 47 below for an example of a keyboard driver that raises the TPL while a check is made to see if a key has been pressed.

4.1.4 Event Services

4.1.4.1 gBS->CreateEvent(), gBS->CreateEventEx, gBS->SetTimer(), gBS->SignalEvent(), and gBS->CloseEvent()

The **gBS->CreateEvent()** and **gBS->CloseEvent()** services are used to create and destroy events. The following two basic types of events can be created:

- **EVT_NOTIFY_SIGNAL**
- **EVT_NOTIFY_WAIT**

The type of event determines when an event's notification function is invoked. The notification function for signal type events are invoked when the event is placed into the signaled state with a call to **gBS->SignalEvent()**. The notification function for wait type events are invoked when the event is passed to the **gBS->CheckEvent()** or **gBS->WaitForEvent()** services.

Some UEFI drivers need to place their controllers in a quiescent state or perform other controller-specific actions at the time that an operating system is about to take full control of the platform. In this case, the UEFI driver should create a signal type event that is notified when **gBS->ExitBootServices()** is called by the operating system. The notification function for this type of event is not allowed to use any of the UEFI Memory Services either directly or indirectly because using those services may modify the memory map, which will force an error to be returned from **gBS->ExitBootServices()**. The code fragment in Example 45 shows how an Exit Boot Services event can be created and destroyed along with the skeleton of the notification function that is invoked when the Exit Boot Services event is signaled. This event is automatically signaled by UEFI firmware when **gBS->ExitBootServices()** is called by an OS loader

or an OS kernel. The notification function that is registered in **gBS->CreateEvent()** is called when the event is signaled.

```
VOID
EFIAPI
NotifyExitBootServices (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  //
  // Put driver-specific actions here.
  // No EFI Memory Service may be used directly or indirectly.
  //
}

EFI_STATUS  Status;
EFI_EVENT   *ExitBootServicesEvent;

//
// Create an Exit Boot Services event.
//
Status = gBS->CreateEvent (
                EFI_EVENT_SIGNAL_EXIT_BOOT_SERVICES,
                EFI_TPL_NOTIFY,
                NotifyExitBootServices,
                NULL,
                &ExitBootServicesEvent
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Destroy the Exit Boot Services event
//
Status = gBS->CloseEvent (ExitBootServicesEvent);
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 45. Exit Boot Services Event

EFI runtime drivers may need to be notified when the operating system sets up virtual mappings for all the UEFI runtime mapping. In this case, the UEFI driver should create a signal type event that is notified when **gBS->SetVirtualAddressMap()** is called by the operating system. This call allows the UEFI runtime driver to convert pointers from physical addresses to virtual addresses. The notification function for this type of event is not allowed to use any of the EFI Boot Services, UEFI console Services, or UEFI protocol Services either directly or indirectly because those services are no longer available when **gRT->SetVirtualAddressMap()** is called. Instead, this type of notification function typically uses **gRT->ConvertPointer()** to convert pointers within data structures that are managed by the UEFI runtime driver from physical addresses to virtual addresses. The code fragment in Example 46 shows how a Set Virtual Address Map event can be created and destroyed along with the skeleton of the notification function that is invoked when the Set Virtual Address Map event is signaled. As an example, this notification function converts a single global pointer from a physical address to a virtual address. This event is automatically signaled by UEFI firmware when **gRT->SetVirtualAddressMap()** is called by an OS loader or an OS

kernel. The notification function that is registered in **gBS->CreateEvent()** is called when the event is signaled.

```
VOID  *gGlobalPointer;

VOID
EFIAPI
NotifySetVirtualAddressMap (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  gRT->ConvertPointer (
         EFI_OPTIONAL_POINTER,
         (VOID **)&gGlobalPointer
         );
}

EFI_STATUS  Status;
EFI_EVENT   *SetVirtualAddressMapEvent;

//
// Create a Set Virtual Address Map event.
//
Status = gBS->CreateEvent (
                EFI_EVENT_SIGNAL_SET_VIRTUAL_ADDRESS_MAP,
                EFI_TPL_NOTIFY,
                NotifySetVirtualAddressMap,
                NULL,
                &SetVirtualAddressMapEvent
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Destroy the Set Virtual Address Map event
//
Status = gBS->CloseEvent (SetVirtualAddressMapEvent);
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 46. Set Virtual Address Map Event

Some UEFI drivers need to be notified on a periodic basis to poll a device. A good example is the USB bus driver, which needs to check the status of all the ports periodically to see if a USB device has been added or removed. Other drivers may need to be notified once after a specific period of time. A good example of this second scenario is the ISA floppy driver than needs to turn off the floppy drive motor if there are no read or write operations for a few seconds. These UEFI drivers would use the **gBS->CreateEvent()** and **gBS->CloseEvent()** services to create and destroy the event and the **gBS->SetTimer()** to arm a periodic timer or a one-shot timer. See section 4.4.2 for examples on how to create periodic timer events and one-shot timer events.

Wait events are the last type of event that will be discussed here. Wait events are typically produced by protocols that abstract I/O services for controllers that provide input services. These protocols will contain an **EFI_EVENT** data field that is signaled by the UEFI driver when the input is available. The following protocols in the *UEFI 2.0 Specification* use this mechanism:

- **EFI_SIMPLE_TEXT_INPUT_PROTOCOL** (section 11.2 in the *UEFI 2.0 Specification*)
- **EFI_SIMPLE_POINTER_PROTOCOL** (section 11.4 in the *UEFI 2.0 Specification*)
- **EFI_SIMPLE_NETWORK_PROTOCOL** (section 20.1 in the *UEFI 2.0 Specification*)

The code fragment in Example 47 shows an example of a wait event that is created by a keyboard driver that produces the **EFI_SIMPLE_TEXT_INPUT_PROTOCOL**. The first part of the code fragment is the event notification function that is called when the wait event is waited on with the **gBS->CheckEvent()** or the **gBS->WaitForEvent()** services. The second part of the code fragment is the code that would be in the **Start()** and **Stop()** functions that create and destroy the wait event. Typically, an UEFI application or the UEFI boot manager will call **gBS->CheckEvent()** or **gBS->WaitForEvent()** to see if a key has been pressed on a input device that supports the **EFI_SIMPLE_TEXT_INPUT_PROTOCOL**. This call to **gBS->CheckEvent()** or **gBS->WaitForEvcent()** will cause the notification function of the wait event in the **EFI_SIMPLE_TEXT_INPUT_PROTOCOL** to be executed. The notification function checks to see if a key has been pressed on the input device. If the key has been pressed, then the wait event is signaled with a call to **gBS->SignalEvent()**. If the wait event is signaled, then the UEFI application or UEFI boot manager will get an **EFI_SUCCESS** return code, and the UEFI application or UEFI boot manager can call the **ReadKeyStroke()** service of the **EFI_SIMPLE_TEXT_INPUT_PROTOCOL** to read the key that was pressed.

```
VOID
EFIAPI
KeyboardWaitForKey (
  IN  EFI_EVENT  Event,
  IN  VOID       *Context
  )
{
  EFI_TPL  OldTpl;

  //
  // Raise the Task Priority Level to EFI_TPL_NOTIFY to perform blocking
I/O
  //
  OldTpl = gBS->RaiseTPL (EFI_TPL_NOTIFY);

  //
  // Call an internal function to see if a key has been pressed
  //
  if (!EFI_ERROR (KeyboardCheckForKey (Context))) {
    //
    // If a key has been pressed, then signal the wait event
    //
    gBS->SignalEvent (Event);
  }

  //
  // Restore the Task Priority Level to its original level
  //
  gBS -> RestoreTPL (OldTpl);

  return;
}

EFI_STATUS                    Status;
EFI_SIMPLE_TEXT_INPUT_PROTOCOL  *SimpleInput;

//
// Create a wait event for a Simple Input Protocol
//
Status = gBS->CreateEvent (
                EFI_EVENT_NOTIFY_WAIT,
                EFI_TPL_NOTIFY,
                KeyboardWaitForKey,
                &SimpleInput,
                &SimpleInput.WaitForKey
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Destroy the wait event
//
Status = gBS->CloseEvent (SimpleInput.WaitForKey);
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 47. Wait for Event

When signaling an event group, it is possible to create an event in the group, signal it and then close the event to remove it from the group. For example:

```
EFI_EVENT   Event;
EFI_GUID    gMyEventGroupGuid = EFI_MY_EVENT_GROUP_GUID;

Status = gBS->CreateEventEx (
                0,
                0,
                NULL,
                NULL,
                &gMyEventGroupGuid,
                &Event
                );

gBS->SignalEvent (Event);
gBS->CloseEvent (Event);
```

Example 48. Signal an Event group

4.1.4.2 gBS->CheckEvent()

This service checks to see if an event is in the waiting state or the signaled state. UEFI drivers will typically use this service to see if a one-shot timer event or a wait event is in the signaled state. For example, the PXE base code driver and the terminal driver create a one-shot timer event and they both use this service to see if a transaction has timed out. The console splitter driver uses this service to see if a key has been pressed or a pointer device has been moved on one of the input devices that it is managing. The code fragment in Example 49 creates a one-shot timer event and uses **gBS->CheckEvent()** to wait until the timer expires.

```
EFI_STATUS  Status;
EFI_EVENT   TimerEvent;

Status = gBS->CreateEvent (
                EFI_EVENT_TIMER | EFI_EVENT_NOTIFY_WAIT,
                EFI_TPL_NOTIFY,
                NULL,
                NULL,
                &TimerEvent
                );

Status = gBS->SetTimer (
                TimerEvent,
                TimerRelative,
                40000000          // 4 seconds in the future
                );

do {
  Status = gBS->CheckEvent (TimerEvent);
} while (EFI_ERROR (Status));
```

Example 49. Wait for a One-Shot Timer Event

4.1.5 Delay Services

4.1.5.1 gBS->Stall()

This service waits for a specified number of microseconds. The range of supported delays is from 1 µS to 4294 seconds. However, the delays passed into this service

should be short and are typically in the range of a few microseconds to a few milliseconds. See section 4.4.2 for examples of how this service can be used.

4.2 Services That UEFI drivers Rarely Use

Table 17 contains the list of UEFI services that are rarely used by UEFI drivers. The following sections provide a brief description of each service, why they are rarely used, and a code example on how they are typically used by UEFI drivers. This table is also present in Appendix C for reference purposes.

Table 17. UEFI Services That Are Rarely Used by UEFI drivers

Type	Service	Type	Service
BS	gBS->ReinstallProtocolInterface()	BS	gBS->LoadImage()
BS	gBS->LocateDevicePath()	BS	gBS->StartImage()
		BS	gBS->UnloadImage()
BS	gBS->ConnectController()	BS	gBS->Exit()
BS	gBS->DisconnectController()		
		BS	gBS->InstallConfigurationTable()
RT	gRT->GetVariable()		
RT	gRT->SetVariable()	RT	gRT->GetTime()
RT	gRT->QueryVariableInfo()		
BS	gBS->GetNextMonotonicCount()	BS	gBS->CalculateCrc32()
RT	gRT->GetNextHighMonotonicCount()		
		RT	gRT->ConvertPointer()

4.2.1 Handle Database and Protocol Services

4.2.1.1 gBS->ReinstallProtocolInterface()

This service should be used only to indicate media change events and when a device path is modified or updated. This service applies to the following:

- The Block I/O Protocol when the media in a removable media device is changed
- The Serial I/O Protocol when its attributes are modified with a call to **`SetAttributes()`**
- The Simple Network Protocol when the MAC address of the network interface is modified with a call to **`StationAddress()`**

This service is basically a series of the following calls, in the order listed:

UninstallProtocolInterface(), which may cause **DisconnectController()** to be called

InstallProtocolInterface()

ConnectConroller() to allow controllers that had to release the protocol a chance to connect to it again

The code fragment in Example 50 shows what an UEFI driver that produces the Block I/O Protocol would do when the media in a removable media device is changed. The exact same protocol is reinstalled onto the controller handle.

```
EFI_STATUS              Status;
EFI_HANDLE              ControllerHandle;
EFI_BLOCK_IO_PROTOCOL   *BlockIo;

Status = gBS->ReinstallProtocolInterface (
                ControllerHandle,
                &gEfiBlockIoProtocolGuid,
                BlockIo,
                BlockIo
                );
```

Example 50. Reinstall Protocol Interface

Note: *This service can cause reentrancy problems if not handled correctly. If a driver makes a request that requires a protocol of a parent device to be updated, then that protocol will be removed and reattached. The driver making the request may not realize that the request will cause the driver to be completely stopped and completely restarted when the request to the parent device is made. For example, consider a terminal driver that wants to change the baud rate on the serial port. The baud rate is changed with a call to the Serial I/O Protocol's **SetAttribute()**. This call changes the baud rate, which is reflected in the device path of the serial device, so the Device Path Protocol is reinstalled by the **SetAttributes()** service. This reinstallation will force the terminal driver to be disconnected. The terminal driver will then attempt to connect to the serial device again, but the baud rate will be the one that the terminal driver expects, so the terminal driver will not need to set the baud rate again. Any consumer of a protocol that supports this media change concept needs to be aware that the protocol can be reinstalled at any time and care must be taken in the design of drivers that use this type of protocol.*

4.2.1.2 gBS->LocateDevicePath()

This service locates a device handle that supports a specific protocol and has the closest matching device path. This service is useful when an UEFI driver needs to find an I/O abstraction for one of its parent controllers. Normally, an UEFI driver will use the services on the *ControllerHandle* that is passed into the **Supported()** and **Start()** functions of the UEFI driver's **EFI_DRIVER_BINDING_PROTOCOL**. However, if an UEFI driver needs to use services from a parent controller, this function can be used to find the handle of a parent controller. For example, a PCI device driver will normally use the PCI I/O Protocol to manage a PCI controller. If the PCI device driver needs the services of the PCI Root Bridge I/O Protocol of which the PCI controller is a child, then the **gBS->LocateDevicePath()** function can be used to find the parent handle that supports the PCI Root Bridge I/O Protocol, and then the **gBS->OpenProtocol()** service can be used to retrieve the PCI Root Bridge I/O Protocol interface. This operation is not

recommended because a parent bus driver typically owns the parent I/O abstractions. Directly using a parent I/O may cause unintended side effects. The code fragment in Example 51 demonstrates this example.

Section 14.4.2 contains another example that shows the recommended method for a PCI driver to access the resources of different PCI controllers without using the PCI Root Bridge I/O Protocol.

```
EFI_STATUS                          Status;
EFI_GUID                            gEfiDevicePathProtocolGuid;
EFI_GUID                            gEfiPciRootBridgeIoProtocolGuid;
EFI_DRIVER_BINDING_PROTOCOL         *This;
EFI_HANDLE                          ControllerHandle;
EFI_DEVICE_PATH_PROTOCOL            *DevicePath;
EFI_HANDLE                          ParentHandle;
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL     *PciRootBridgeIo;

//
// Retrieve the Device Path Protocol instance on ControllerHandle
//
Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiDevicePathProtocolGuid,
                &DevicePath,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
if (EFI_ERROR (Status)) {
  return Status;
}


//
// Find a parent controller that supports the PCI Root Bridge I/O
Protocol
//
Status = gBS->LocateDevicePath (
                &gEfiPciRootBridgeIoProtocolGuid,
                &DevicePath,
                &ParentHandle
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Get the PCI Root Bridge I/O Protocol instance on ParentHandle
//
Status = gBS->OpenProtocol (
                ParentHandle,
                &gEfiPciRootBridgeIoProtocolGuid,
                &PciRootBridgeIo,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 51. Locate Device Path

4.2.1.3 gBS->ConnectController() and gBS->DisconnectController()

These services are used by UEFI drivers that are bus drivers for bus types with hot-plug capabilities that are supported in the preboot environment. The only bus driver in the *EDK* that uses these services is the USB bus driver. The USB bus driver does not create any child handles in its **Start()** function. Instead, it registers a periodic timer event. Each time the timer period expires, the timer event's notification function is called, and that notification function examines all the USB hubs to see if any USB devices have been added or removed. If a USB device has been added, then a child handle is created, and **gBS->ConnectController()** is called so the USB device drivers can connect to the newly added USB device. If a USB device has been removed, then **gBS->DisconnectController()** is called to stop the USB device drivers from managing the USB device that was just removed. Just because a bus is capable of supporting hot-plug events does not necessarily mean that the UEFI driver for that bus type must support those hot-plug events. Support for hot-plug events in the preboot environment is dependent on the platform requirements for each bus type. The code fragment in Example 52 shows how these services could be used from the USB bus driver.

```
EFI_STATUS  Status;
BOOLEAN     HotAdd;
BOOLEAN     HotRemove;
EFI_HANDLE  ChildHandle;

//
// If ChildHandle is a device that was just hot added, then recursively
// connect all drivers to ChildHandle
//
if (HotAdd == TRUE) {
  Status = gBS->ConnectController(
                  ChildHandle,
                  NULL,
                  NULL,
                  TRUE
                  );
}

//
// If ChildHandle is a device that was just removed, then recursively
// disconnect all drivers from ChildHandle
//
if (HotRemove == TRUE) {
  Status = gBS->DisconnectController(
                  ChildHandle,
                  NULL,
                  NULL
                  );
}
```

Example 52. Connect and Disconnect Controller

The **gBS->DisconnectController()** service may also be used from unloadable UEFI drivers to disconnect the UEFI driver from the device it is managing in its **Unload()** function. The code fragment in Example 53 shows one algorithm that an **Unload()** function can use to disconnect the driver from all the devices in the system. It retrieves the list of all the handles in the handle database and disconnects the UEFI driver from each of those handles and frees the buffer containing the list handles.

```
EFI_STATUS  Status;
EFI_HANDLE  ImageHandle;
EFI_HANDLE  *DeviceHandleBuffer;
UINTN       DeviceHandleCount;
UINTN       Index;

Status = gBS->LocateHandleBuffer (
                AllHandles,
                NULL,
                NULL,
                &DeviceHandleCount,
                &DeviceHandleBuffer
                );
if (EFI_ERROR (Status)) {
  return Status;
}

for (Index = 0; Index < DeviceHandleCount; Index++) {
  Status = gBS->DisconnectController (
                  DeviceHandleBuffer[Index],
                  ImageHandle,
                  NULL
                  );
}

gBS->FreePool(DeviceHandleBuffer);
```

Example 53. Disconnect Controller

4.2.2 Image Services

4.2.2.1 gBS->LoadImage(), gBS->StartImage(), gBS->UnloadImage(), and gBS->Exit()

UEFI drivers should not call **gBS->Exit()**. Instead, they should just return a status code from their driver entry point.

UEFI drivers do not normally load and unload other UEFI drivers or UEFI applications. However, there are two exceptions. The first exception is that bus drivers that can load, start, and potentially unload UEFI drivers that are stored in containers on the child devices of the bus. For example, the PCI bus driver loads and starts UEFI drivers that are stored in PCI option ROMs.

The second exception is for devices that have an UEFI driver that manages the device and an UEFI application that provides the user interface that is used to configure the device. In this case, the implementation of the **SetOptions()** service in the **EFI_DRIVER_CONFIGURATION_PROTOCOL** uses the **gBS->LoadImage()** and **gBS->StartImage()** services to load and execute the UEFI application to configure the device.

4.2.3 Variable Services

4.2.3.1 gRT->GetVariable() and gRT->SetVariable()

These services are used to get and set UEFI variables. When variables are stored, there are attributes that describe the visibility and persistence of the variable. These attributes can be combined different types of variables. The legal combinations of attributes include the following:

BOOTSERVICE_ACCESS The variable is available for read and write access in the preboot environment before **gBS->ExitBootServices()** is called. The variable is not available after **gBS->ExitBootServices()** is called, and contents are also lost on the next system reset or power cycle. These types of variables are typically used to share information among different preboot components.

BOOTSERVICE_ACCESS | RUNTIME_ACCESS
The variable is available for read and write access in the preboot environment before **gBS->ExitBootServices()** is called. It is available for read-only access from the OS runtime environment after **gBS->ExitBootServices()** is called. The contents are lost on the next system reset or power cycle. These types of variable are typically used to share information among different preboot components and pass read-only information to the operating system.

NON_VOLATILE | BOOTSERVICE_ACCESS
The variable is available for read and write access in the preboot environment before **gBS->ExitBootServices()** is called, and the contents are persistent across system resets and power cycles. These types of variables are typically used to share persistent information among different preboot components.

NON_VOLATILE | BOOTSERVICE_ACCESS | RUNTIME_ACCESS
The variable is available for read and write access in both the preboot environment and the OS runtime environment. The contents are persistent across system resets and power cycles. These types of variables are typically used to share persistent information among preboot components and the operating system.

The code fragment in Example 54 shows how a fixed-sized UEFI variable can be read and written. A variable name is a combination of a GUID and a Unicode string. The GUID allows private variables to be managed by different vendors. Section 3.2 of the *UEFI 2.0 Specification* defines one GUID that is used to access UEFI variables. This example shows how the fixed-size UEFI variable *BootNext* can be accessed.

```
EFI_STATUS  Status;
UINT32      Attrubutes;
UINTN       DataSize;
UINT16      BootNext;

DataSize = sizeof (UINT16);
Status = gRT->GetVariable (
                L"BootNext",
                &gEfiGlobalVariableGuid,
                &Attributes,
                &DataSize,
                &BootNext
                );

BootNext = 2;
Status = gRT->SetVariable (
                L”BootNext”,
                &gEfiGlobalVariableGuid,
                EFI_VARIABLE_NON_VOLATILE |
                  EFI_VARIABLE_BOOTSERVICE_ACCESS |
                  EFI_VARIABLE_RUNTIME_ACCESS,
                sizeof (UINT16),
                &BootNext
                );
```

Example 54. Reading and Writing Fixed-Size UEFI variables

The code fragment in Example 55 shows how a memory buffer for the variable-sized UEFI variable *Driver001C* can be allocated. The variable is then read into the allocated buffer and written to the UEFI variable *Driver001D*. Finally, the allocated buffer is freed.

```
EFI_STATUS  Status;
UINT32      Attrubutes;
UINTN       DataSize;
VOID        *Data;

DataSize = 0;
Status = gRT->GetVariable (
                L"Driver001C",
                &gEfiGlobalVariableGuid,
                &Attributes,
                &DataSize,
                Data
                );
if (Status != EFI_BUFFER_TOO_SMALL) {
  return Status;
}

Data = EfiLibAllocatePool (DataSize);
if (Data == NULL) {
  return EFI_OUT_OF_RESOURCES;
}

Status = gRT->GetVariable (
                L"Driver001C",
                &gEfiGlobalVariableGuid,
                &Attributes,
                &DataSize,
                Data
                );
if (EFI_ERROR (Status)) {
  return Status;
}

Status = gRT->SetVariable (
                L"Driver001D",
                &gEfiGlobalVariableGuid,
                EFI_VARIABLE_NON_VOLATILE |
                  EFI_VARIABLE_BOOTSERVICE_ACCESS |
                  EFI_VARIABLE_RUNTIME_ACCESS,
                DataSize,
                Data
                );

gBS->FreePool (Data);
```

Example 55. Reading and Writing Variable-Sized UEFI variables

4.2.4 Time Services

4.2.4.1 gRT->GetTime()

This service is typically only accurate to about 1 second. As a result, UEFI drivers should not use this service to poll or wait for an event from a device. Instead, the **gBS->Stall()** service and the **gBS->SetTimer()** services provide time services with much higher accuracy. This service should be used only when the current time and date are required such as recording the time and date of a critical error. See section 4.4.3 for more details on the time-related services that are provided by EFI.

4.2.5 Virtual Memory Services

4.2.5.1 gRT->ConvertPointer()

This service is never used by UEFI Boot Service drivers and is sometimes used by UEFI runtime drivers. Example 46 shows an example of how a UEFI runtime driver might use this service.

4.2.6 Miscellaneous Services

4.2.6.1 gBS->InstallConfigurationTable()

This service is used to add, update, or remove an entry in the list of configuration tables that is maintained in the EFI System Table. These configuration tables are typically used to pass information from the UEFI environment into an operating system environment. UEFI Boot Service drivers are destroyed at **gBS->ExitBootServices()**, so they do not typically need to pass any information to an operating system. UEFI runtime drivers continue to persist after **gBS->ExitBootServices()**, so they may need to pass information to an operating system so that the operating system can use the services that the UEFI runtime driver produced. Typically, only the class of UEFI runtime drivers that need to pass information to the operating system will use this service. The code fragment in Example 56 shows how an UNDI driver can add, update, and remove a configuration table. The first parameter is a pointer to the Network Interface Identifier (NII) GUID, and the second parameter is a pointer to the data structure that is associated with the NII GUID.

```
EFI_STATUS  Status;
NII_TABLE   *UndiData;

//
// Add or update a configuration table
//
Status = gBS->InstallConfigurationTable (
                &gEfiNetworkInterfaceIdentifierProtocolGuid_31,
                UndiData
                );

//
// Remove a configuration table
//
Status = gBS->InstallConfigurationTable (
                &gEfiNetworkInterfaceIdentifierProtocolGuid_31,
                NULL
                );
```

Example 56. Install Configuration Table

4.2.6.2 gBS->CalculateCrc32()

This service is used by the UEFI core to maintain the checksum of the EFI System Table, EFI Boot Services Table, and EFI Runtime Services Table. It is also used by a few UEFI drivers including the Console Splitter and the Partition driver for Guided Partition Table (GPT) disk. If an UEFI driver requires the use of 32-bit CRCs, the size of

the UEFI driver can be reduced by using this UEFI boot service. The code fragment in Example 57 shows how this service can be used to calculate a 32-bit CRC value for *Size* bytes of a buffer that has a header of type **EFI_TABLE_HEADER**. **EFI_TABLE_HEADER** is defined in section 4.2 of the *UEFI 2.0 Specification*, and it is the standard header used for the EFI System Table, EFI Boot Services Table, EFI Runtime Services Table, and GPT related structures.

```
EFI_STATUS        Status;
UINT32            Crc;
EFI_TABLE_HEADER  *Header;
UINT32            Size;

Header->CRC32 = 0;
Status = gBS->CalculateCrc32(
                (VOID *)Header,
                Size,
                &Crc
                );
Hdr->CRC32 = Crc;
```

Example 57. Computing 32-bit CRC Values

This service can also be used to verify a 32-bit CRC value. The code fragment in Example 58 shows how the 32-bit CRC for a buffer of *Size* bytes with an **EFI_TABLE_HEADER** can be validated. It returns **TRUE** if the 32-bit CRC is good. Otherwise, it returns **FALSE**.

```
EFI_STATUS        Status;
UINT32            OriginalCrc;
UINT32            Crc;
EFI_TABLE_HEADER  *Header;
UINT32            Size;

OriginalCrc = Header->CRC32;
Header->CRC32 = 0;
Status = gBS->CalculateCrc32(
                (VOID *)Header,
                Size,
                &Crc
                );
Hdr->CRC32 = OriginalCrc;

if (OriginalCrc == Crc) {
  return TRUE;
} else {
  return FALSE;
}
```

Example 58. Validating 32-bit CRC Values

4.2.6.3 gBS->GetNextMonotonicCount() and gRT->GetNextHighMonotonicCount()

These services provide a 64-bit monotonic counter that is guaranteed to increase. They are not used by any of the drivers in the *EDK*.

4.3 Services That UEFI drivers Should Not Use

Table 18 lists the UEFI services that should not be used by UEFI drivers. The following sections describe why each of these functions should not be used. This table is also present in Appendix C for reference purposes.

Table 18. UEFI Services That Should Not Be Used by UEFI drivers

Type	Service	Type	Service
BS	gBS->GetMemoryMap()	RT	gRT->SetVirtualAddressMap()
BS	gBS->ExitBootServices()	RT	gRT->GetNextVariableName()
BS	gBS->InstallProtocolInterface()	RT	gRT->SetTime()
BS	gBS->UninstallProtocolInterface()	RT	gRT->GetWakeupTime()
BS	gBS->HandleProtocol()	RT	gRT->SetWakeupTime()
BS	gBS->LocateHandle()		
BS	gBS->RegisterProtocolNotify()	RT	gRT->ResetSystem()
BS	gBS->ProtocolsPerHandle()	BS	gBS->SetWatchDogTimer()
BS	gBS->WaitForEvent()		

4.3.1 Memory Services

4.3.1.1 gBS->GetMemoryMap()

UEFI drivers should not use this service because UEFI drivers should not depend upon the physical memory map of the platform. The **`gBS->AllocatePool()`** and **`gBS->AllocatePages()`** services allow an UEFI driver to allocate system memory. The **`gBS->FreePool()`** and **`gBS->FreePages()`** allow an UEFI driver to free previously allocated memory. If there are limitations on the memory areas that a specific device may use, then those limitations should be managed by a parent I/O abstraction that understands the details of the platform hardware. For example, PCI device drivers should use the services of the PCI I/O Protocol to manage DMA buffers. The PCI I/O Protocol is produced by the PCI bus driver that uses the services if the PCI Root Bridge I/O Protocol to manage DMA buffers. The PCI Root Bridge I/O Protocol is chipset and platform specific, so the component that produces the PCI Root Bridge I/O Protocol understands what memory regions can be used for DMA operations. By pushing the responsibility into the chipset- and platform-specific components, the PCI device drivers and PCI bus drivers are easier to implement and are portable across a wide variety of platforms.

4.3.2 Image Services

4.3.2.1 gBS->ExitBootServices()

This service is used only by UEFI OS loaders or OS kernels. It hands control of the platform from the UEFI compliant firmware to an OS. After this call, the EFI Boot Services are no longer available, and all memory in use by EFI Boot Service drivers is considered to be available memory. If this call is made by an EFI Boot Service driver, it would essentially destroy itself.

4.3.3 Handle Database and Protocol Services

4.3.3.1 gBS->InstallProtocolInterface()

This service adds one protocol interface to an existing handle or creates a new handle. This service has been replaced by the **`gBS->InstallMultipleProtocolInterfaces()`** service, so all UEFI drivers should use the newer service. Using this replacement service provides additional flexibility and additional error checking and produces smaller UEFI drivers.

4.3.3.2 gBS->UninstallProtocolInterface()

This service removes one protocol interface from a handle in the handle database. The functionality of this service has been replaced by **`gBS->UninstallMultipleProtocolInterfaces()`**. This service uninstalls one or more protocol interfaces from the same handle. Using this replacement service provides additional flexibility and produces smaller UEFI drivers.

4.3.3.3 gBS->HandleProtocol()

UEFI drivers must not use this service because they will not be compliant with the UEFI Driver Model if they do, which could introduce interoperability issues. Instead, **gBS->OpenProtocol()** should be used because it provides the equivalent functionality, and it allows the UEFI core to track the agents that are using different protocol interfaces in the handle database.

4.3.3.4 gBS->LocateHandle()

This service returns an array of handles that support the specified protocol. This service requires the caller to allocate the return buffer. The **gBS->LocateHandleBuffer()** service is easier to use and produces smaller executables because it allocates the return buffer for the caller.

4.3.3.5 gBS->RegisterProtocolNotify()

This service registers an event that is to be signaled whenever an interface is installed for a specified protocol. Using this service is strongly discouraged. This service was previously used by EFI drivers that follow the *EFI 1.02 Specification*, and it provided a simple mechanism for drivers to layer on top of another driver. Chapter 9 of the *UEFI 2.0 Specification* instead defines the UEFI Driver Model, which provides a much more flexible mechanism.

4.3.3.6 gBS->ProtocolsPerHandle()

This service retrieves the list of protocols that are installed on a handle. Because UEFI drivers should already know what protocols are installed on the handles that the UEFI driver is managing, this service should not be used. This service is typically used by UEFI applications that need to traverse the entire handle database.

4.3.4 Event Services

4.3.4.1 gBS->WaitForEvent()

This service waits for an event in an event list to be signaled. UEFI drivers are typically waiting only for a single event to enter the signaled state, so the **gBS->CheckEvent()** service should be used instead.

4.3.5 Virtual Memory Services

4.3.5.1 gBS->SetVirtualAddressMap()

This service is also used only by UEFI OS loaders or OS kernels for operating systems that wish to call EFI Runtime Services using virtual addresses. This service must be called after **gBS->ExitBootServices()** is called. As a result, it is not legal for UEFI drivers to call this service.

4.3.6 Variables Services

4.3.6.1 gRT->GetNextVariableName()

This service is used to walk the list of UEFI variables that are maintained through the UEFI variable Services. Most UEFI drivers should already know the UEFI variables they want to access, so there is no need for an UEFI driver to walk the list of all the UEFI variables.

4.3.7 Time Services

4.3.7.1 gRT->SetTime(), gRT->GetWakeupTime(), and gRT->SetWakeupTime()

UEFI drivers should not modify the system time or the wakeup timer. The management of these timer services should be left to the UEFI boot manager, an OEM-provided utility, or an operating system.

4.3.8 Miscellaneous Services

4.3.8.1 gBS->SetWatchdogTimer()

The watchdog timer is managed from the UEFI boot manager, so UEFI drivers should not use this service.

4.3.8.2 gRT->ResetSystem()

System resets should be managed from the UEFI boot manager or OEM-provided utilities. UEFI drivers should not use this service. The only exceptions in the *EDK* are the keyboard drivers that detect the CTRL-ALT-DEL key sequence to reset the platform.

4.4 Time-Related Services

There are several different time-related services that are available to UEFI drivers, and they are listed below in Table 19. The time-related services that should not be used by UEFI drivers are discussed in section 4.4.2 and include the following:

- **`gRT->SetTime()`**
- **`gRT->GetWakeupTime()`**
- **`gRT->SetWakeupTime()`**
- **`gBS->SetWatchDogTimer()`**

Because UEFI drivers should not use these services, they will not be discussed. Omitting the four services above leaves the following three time-related services that UEFI drivers can use:

- **gBS->SetTimer()**
- **gBS->Stall()**
- **gRT->GetTime()**

The **gBS->SetTimer()** and **gBS->Stall()** services are commonly used, but the **gRT->GetTime()** service is rarely used.

If an UEFI driver requires highly accurate short delays, then the **gBS->Stall()** service should be used. If an UEFI driver needs to synchronize periodically with a device, then the **gBS->SetTimer()** service should be used with an event. If an UEFI driver needs to know the current time and date, then the **gRT->GetTime()** service should be used.

Table 19. Time-Related UEFI Services

Type	Service	Type	Service
BS	gBS->SetTimer()	RT	gRT->GetWakeupTime()
BS	gBS->Stall()	RT	gRT->SetWakeupTime()
RT	gRT->GetTime()	BS	gBS->SetWatchDogTimer()
RT	gRT->SetTime()		

4.4.1 Stall Service

The **gBS->Stall()** service is the time-related service with the highest accuracy. The stall times can range from 1 µS to about 4294 seconds. Most implementations of the stall service use a calibrated software loop, so they are very accurate. This service is a good one to use when an UEFI driver requires a short delay. For example, an UEFI driver may send a command to a controller and then wait for the command to complete. Because all UEFI drivers are polled, the UEFI driver could use the stall service inside a loop to check periodically for the completion status (see Example 59). Another good use of the stall service is for hardware devices that require delays between register accesses. Here, a fixed stall value would be used, and the stall value would be based in a hardware specification for the device that is being accessed (see Example 60). One disadvantage of the stall service is that other UEFI components cannot execute while a stall is being executed. If long delays are required and it makes sense to defer the completion of an I/O operation, then the timer event services described in the next section should be used.

```
EFI_STATUS           Status;
UINTN                TimeOut;
EFI_PCI_IO_PROTOCOL  PciIo;
UITN8                Value;

//
// Loop waiting for the register at Offset 0 of Bar #0 of PciIo to become
0xE0.
// Wait 10 uS between each check of this register, and time out if it
does
// not become 0 after 100 mS.
//
TimeOut = 0;
do {
  //
  // Wait 10 uS
  //
  gBS->Stall(10);

  //
  // Increment TimeOut by the number of stalled uS
  //
  TimeOut = TimeOut + 10;

  //
  // Do a single 8 bit read from BAR #0, Offset 0 into Value
  //
  Status = PciIo->Io.Read (
                       PciIo,                // This
                       EfiPciIoWidthUint8,   // Width
                       0,                    // BarIndex
                       0,                    // Offset
                       1,                    // Count
                       &Value                // Buffer
                       );
} while (Value != 0xE0 && TimeOut <= 100000);

if (Value != 0xE0) {
  return EFI_TIMEOUT;
}

return EFI_SUCCESS;
```

Example 59. Stall Loop

```
EFI_STATUS           Status;
EFI_PCI_IO_PROTOCOL  PciIo;
UITN8                Value;

//
// This example shows a stall of 1000 uS between two register writes to
the
// same register. The 1000 uS is based on a hardware requirement for the
// device being accessed.
//

//
// Do a single 8 bit write to BAR #1, Offset 0x10 of 0xAA
//
Value = 0xAA;
Status = PciIo->Io.Write (
                     PciIo,                // This
                     EfiPciIoWidthUint8,   // Width
                     1,                    // BarIndex
                     0x10,                 // Offset
                     1,                    // Count
                     &Value                // Buffer
                     );

//
// Wait 1000 uS
//
gBS->Stall(1000);

//
// Do a single 8-bit write to BAR #1, Offset 0x10 of 0x55
//
Value = 0x55;
Status = PciIo->Io.Write (
                     PciIo,                // This
                     EfiPciIoWidthUint8,   // Width
                     1,                    // BarIndex
                     0x10,                 // Offset
                     1,                    // Count
                     &Value                // Buffer
                     );
```

Example 60. Stall Service

4.4.2 Timer Events

The timer event services allow an UEFI driver to wait for a specified period of time before a notification function is called. This service is a good choice for UEFI drivers that need to defer the completion of an I/O operation, thus allowing other UEFI components to continue to execute while the UEFI driver waits for the specified period of time. The time is specified in 100 nS units. This unit may give the appearance of having better accuracy than the **`gBS->Stall()`** service, which has an accuracy of 1 µS, but that is not the case. The UEFI core uses a single timer interrupt to determine when to signal timer events. The resolution of timer events is completely dependent on the frequency of the timer interrupt that is used by the UEFI core. In most systems, the timer interrupt is generated every 10 to 50 mS, but the *UEFI 2.0 Specification* does not require any specific interrupt rate. This lack of specificity means that a periodic timer that is set with a 100 nS period will actually get called only every 10 mS to 50 mS, which is why the **`gBS->Stall()`** service is a much better choice for short delays.

However, there are cases when timer events work very well. One example is the USB bus driver. This driver is required to check periodically the status of all the USB hubs to see if any USB devices have been attached or removed. This operation does not need to be performed very frequently, so the timer events work very well. Example 61 shows how to set up a periodic timer event with a period of 100 mS. This period is similar to what the USB bus driver would use. Another good example is a floppy driver. This driver needs to manage the drive motor on the floppy drive so that it is automatically turned off if there are no floppy transactions for a period of time. Example 62 shows how to set up a timer event that will fire 4 seconds in the future. If a timer like this is rearmed every time a floppy transaction is performed, then the notification function for this timer could be used to turn off the floppy drive motor.

```
//
// This is the notification function used in the peridic timer example
below.
// An UEFI driver will perform a driver-specific operation inside this
function
// The Context parameter will typically contain a private context
structure for
// a device that is managed by the UEFI driver.
//
VOID
TimerHandler (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )

{
}

EFI_STATUS  Status;
EFI_EVENT   TimerEvent;
VOID        *Context;

//
// Create a timer event that will call the notification function
TimerHandler()
// at a TPL level of EFI_TPL_NOTIFY when the timer fires. A context can
be
// passed into the notification function when the timer fires, and that
context
// is specified by the VOID pointer Context. When the event is created,
it is
// returned in TimerEvent.
//
Status = gBS->CreateEvent (
                EFI_EVENT_TIMER | EFI_EVENT_NOTIFY_SIGNAL,
                EFI_TPL_NOTIFY,
                TimerHandler,
                Context,
                &TimerEvent
                );

//
// Set the timer value for the event created above to be a periodic timer
with
// a period of 100 mS. The timer values are specified in 100 nS units.
// The notification function TimerHandler() will be called every 100 mS,
and
// Context will be passed to the notification function.
//
Status = gBS->SetTimer (
                TimerEvent,
                TimerPeriodic,
                1000000           // Every 100 mS
                );
```

Example 61. Starting a Periodic Timer

```
//
// This is the notification function used in the one-shot timer example
below.
// An UEFI driver will perform a driver-specific operation inside this
function.
// The Context parameter will typically contain a private context
structure for
// a device that is managed by the UEFI driver.
//
VOID
TimerHandler(
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )

{
}

EFI_STATUS  Status;
EFI_EVENT   TimerEvent;
VOID        *Context;

//
// Create a timer event that will call the notification function
TimerHandler()
// at a TPL level of EFI_TPL_NOTIFY when the timer fires. A context can
be
// passed into the notification function when the timer fires, and that
context
// is specified by the VOID pointer Context. When the event is created,
it is
// returned in TimerEvent.
//
Status = gBS->CreateEvent (
                EFI_EVENT_TIMER | EFI_EVENT_NOTIFY_SIGNAL,
                EFI_TPL_NOTIFY,
                TimerHandler,
                Context,
                &TimerEvent
                );

//
// Set the timer value for the event created above to be a one-shot timer
with
// a delay of 4 seconds. The timer values are specified in 100 nS units.
// The notification function TimerHandler() will be called 4 seconds
after this
// call returns, and Context will be passed to the notification function.
//
Status = gBS->SetTimer (
                TimerEvent,
                TimerRelative,
                40000000          // 4 seconds in the future
                );
```

Example 62. Arming a One-Shot Timer

Once an UEFI driver is finished with an event timer, the timer must be stopped. The **gBS->SetTimer()** service can be used to cancel the event timer, and the **gBS->CloseEvent()** service can be used to free the event that was allocated in **gBS->CreateEvent()**. If the timer was created in the **EFI_DRIVER_BINDING_PROTOCOL.Start()** function, then this operation would be performed in the **EFI_DRIVER_BINDING_PROTOCOL.Stop()** function. If the timer was created in the driver's entry point, then this operation would be performed in the

driver's unload function if the driver has an unload function. Example 63 below shows how to cancel and free the timer event that was created for the examples in Example 61 and Example 62.

```
EFI_STATUS  Status;
EFI_EVENT   TimerEvent;

Status = gBS->SetTimer (
                TimerEvent,
                TimerCancel,
                0
                );

Status = gBS->CloseEvent (
                TimerEvent
                );
```

Example 63. Stopping a Timer

4.4.3 Time and Date Services

Example 64 shows two examples of the **gRT->GetTime()** service. The first retrieves the current time and date in an **EFI_TIME** structure and it retrieves the capabilities of the real-time clock hardware in an **EFI_TIME_CAPABILITIES** structure. The second example just retrieves the current time and date in an **EFI_TIME** structure and does not retrieve the capabilities of the real time clock hardware. See the U*EFI 2.0 Specification* for a detailed description of this service and its associated data structures.

```
EFI_STATUS             Status;
EFI_TIME               Time;
EFI_TIME_CAPABILITIES  TimeCapabilities;

Status = gRT->GetTime (
                &Time,
                &TimeCapabilities
                );

Status = gRT->GetTime (
                &Time,
                NULL       // Capabilities is optional and may be NULL
                );
```

Example 64. Time and Date Services

4.5 GlueLib Driver Library

The GlueLib is a very useful set of driver functions that has 2 main benefits for any driver developer.

- These functions are well optimized and as such will perform actions quickly
- These functions will work in both EDK and EDK II so porting code between the two build environments is easier.

Gluelib functions are all optimized at least as well as the corresponding EFI Driver Library function. There is no reason to use the EFI Driver Library function over the GlueLib functions.

4.6 UEFI driver Library

Table 58 in Appendix C contains the list of UEFI driver Library Services that are available to UEFI drivers. The details of these services can be found in the EFI Driver Library Specification. These library services aid in the development of UEFI drivers in several ways:

- These library services have been tested in a wide variety of driver types, so they are well validated.
- The UEFI driver Library is a lightweight library that helps reduce driver size compared to other UEFI libraries.
- The UEFI driver Library is designed to complement the services that are already provided by the EFI Boot Services Table, the EFI Runtime Services Table, and the various protocol services. By using the combination of all of these services and the UEFI driver Library, the driver writer can concentrate on implementing the code that directly applies to the device being managed.

5 General Driver Design Guidelines

This chapter contains general guidelines for the design of all types of UEFI drivers. Specific guidelines for specific driver types (PCI drivers, USB drivers, SCSI drivers, etc…) are presented in later chapters.

5.1 General Porting Considerations

Some UEFI drivers are ported from IA-32 BIOS designs or older EFI drivers. The process of porting a driver from one environment to another is often done to save time and leverage resources. If the port is done before obtaining a complete understanding of the target environment, the final driver may have remnants of the previous design that may not belong in the new environment. This section describes the practical porting issues (design and coding) that should be carefully reviewed before porting a driver.

5.1.1 How to Implement Features in EFI

The first column of Table 20 describes functions that a typical driver performs. The second column briefly describes how this function is implemented in EFI and references the section in this guide that specifically addresses each issue. This list of driver operations is not exhaustive.

Table 20. Mapping Operations to UEFI drivers

Operation	Recommended EFI Method
Find devices that the driver supports	Do not search. An UEFI driver is passed a controller handle to evaluate along with a partial device path. See Chapter 9.
Perform DMA	Use the DMA-related services from the PCI I/O Protocol. See section 14.5.
Access PCI configuration header	Use PCI I/O Protocol services. Never directly access I/O ports 0xCF8 or 0xCFC. See Chapter 14 and section 20.2.
Access PCI I/O ports	Use PCI I/O Protocol services. Never use **IN** or **OUT** instructions. See Chapter 14 and section 20.2.
Access PCI memory	Use PCI I/O Protocol services. Never use pointers to directly access memory-mapped I/O resources on a bus. See Chapter 14 and section 20.2.

Hardware interrupts	EFI does not support hooking interrupts. Instead, UEFI drivers are expected to either perform block I/O where they must complete their I/O operation and poll their device as required to complete it, or they can create a periodic timer event to periodically get control to check the status of the devices it is managing. See section 4.4.2 and section 5.6.
Calibrated stalls	Do not use hardware devices to perform calibrated stalls. Instead, use the **gBS->Stall()** service for short delays that are typically less than 10 mS, and one-shot timer events for long delays that are typically greater than 10 mS. The **gRT->GetTime()** service should not be used for delays in UEFI drivers. See section 4.4.1.
Get keyboard input from user	Use console services from within the **SetOptions()** service of the **EFI_DRIVER_CONFIGURATION_PROTOCOL**. Do not use console services anywhere else in the driver because console services may not be available when the driver executes. See chapter 11.
Display text	Use console services from within the **SetOptions()** service of the **EFI_DRIVER_CONFIGURATION_PROTOCOL**. Do not print messages from anywhere else in the driver because the console services may not be available when the driver executed. The **DEBUG()** macro can be used to send debug messages to the standard error console device. See chapter 11 and section section .
Diagnostics	Implement the **EFI_DRIVER_DIAGNOSTICS_PROTOCOL**. See chapter 12.
Flash utility	Write a UEFI utility (stand alone application) to reprogram the flash device.
Prepare controllers for use by an OS	The OS-present drivers should not make assumptions about the state of a controller. It should not assume that the controller was touched by an UEFI driver before the OS was booted. If a specific state is required, then the driver can use an Exit Boot Services event to put the controller into the required state. See section 7.1.3.

5.2 Design and Implementation of UEFI drivers

The following is the list of basic steps that a driver writer should follow when designing and implementing an UEFI driver. Note that this document assumes UEFI Driver Model drivers are being developed.

1. Determine the class of UEFI driver that needs to be developed. The different classes are listed in Table 22 below and are described in more detail in chapter 6 of this document.

Table 21. Classes of UEFI drivers to Develop

Class of Driver	For more information, see sections…
Device driver	6.1 and 9
Bus driver that can produce one or all child handles [Note 1]	6.2, 6.2.6, and 9
Bus driver that produces all child handles in the first call to **`Start()`**	6.2, 6.2.7, and 9
Bus driver that produces at most one child handle in **`Start()`**	6.2, 6.2.8, and 9
Bus driver that produces no child handles in **`Start()`**	6.2, 6.2.9, and 9
Bus driver that produces child handles with multiple parent controllers	6.2, 6.2.10, and 9
Hybrid driver that can produce one or all child handles [Note 1]	6.3, 6.2.6, and 9
Hybrid driver that produces all child handles in the first call to **`Start()`**	6.3, 6.2.7, and 9
Hybrid driver that produces at most one child handle in **`Start()`**	6.3, 6.2.8, and 9
Hybrid driver that produces no child handles in **`Start()`**	6.3, 6.2.9, and 9
Hybrid driver that produces child handles with multiple parent controllers	6.3, 6.2.10, and 9
Service driver	6.4 and 7.3
Root bridge driver	6.5 and 7.4
Initializing driver	6.6 and 7.2

Note:

2. This driver type is recommended because it will enable faster boot times.
3. Is the UEFI driver going to support unloading? See section 7.1.2. This feature is strongly recommended for all drivers.
4. Is the UEFI driver going to produce the Component Name Protocol? See chapter 10. This step is strongly recommended for all drivers.
5. Is the UEFI driver going to produce the Driver Configuration Protocol? See chapter 11. This step is required if the driver will be configured by a user.
6. Is the UEFI driver going to produce the Driver Diagnostics Protocol? See chapter 12. This step is required if the driver will support any testing.
7. If the UEFI driver is a bus driver for a bus type that supports storage of UEFI drivers with the child devices, then the Bus Specific Driver Override Protocol must be implemented by the bus driver. See chapter 13.
8. Is the UEFI driver going to require an Exit Boot Services event? See section 7.1.3.
9. Is the UEFI driver a runtime driver? See section 7.5 and section 23.

10. Is the UEFI driver going to require a Set Virtual Address Map event? See section 7.5.
11. Identify the I/O-related protocols that the driver needs to consume. Based on the list of consumed protocols and the criteria for these protocol interfaces, determine how many instances of the Driver Binding Protocol need to be produced. See sections 6.1.6 and 6.2.5.
12. Identify the I/O-related protocols that the driver needs to produce. All device drivers, bus drivers, and hybrid drivers will use this method.
13. Implement the driver's entry point. See chapter 7.
14. Design the private context data structure. See chapter 8.
15. Implement all the services of **`EFI_DRIVER_BINDING_SERVICES`**. See chapter 9.
16. Implement functions for each produced protocol. See the *UEFI 2.0 Specification* for details on the protocols that are being produced.
17. Is the UEFI driver for a PCI-related device? See chapter 14.
18. Is the UEFI driver for a USB-related device? See chapter 15.
19. Is the UEFI driver for a SCSI-related device? See chapter 16.
20. Is the UEFI driver for a LAN-related device? See chapter 17.
21. Is the UEFI driver for a Graphics-related device? See chapter18.
22. Is the UEFI driver going to see a native driver for the Itanium processor? See chapter 21.
23. Is the UEFI driver going to be an EBC driver? See chapter 22.
24. Design for maximum portability. See the porting considerations for Itanium architecture and EBC in chapter 21 and chapter 22.
25. Design for small code size and improved performance. See chapter 20.

5.3 Maximize Platform Compatibility

UEFI drivers should make as few assumptions about a system's architecture as possible. Minimizing the number of assumptions will maximize the UEFI driver's platform compatibility. It will also reduce the amount of driver maintenance when a driver is deployed on new systems.

5.3.1 Do Not Make Assumptions about System Memory Configurations

Do not make any assumptions about the system memory configuration, including memory allocations and memory that is used for DMA buffers. There may be unexpected gaps in the memory map, and entire memory regions may be missing.

EFI is designed for a wide variety of platforms. As such, portable drivers should not have artificial hard-coded limits; instead, they should rely on published specifications, EFI, and the system firmware to provide them with the platform limitations and platform resources, including the following:

- The number of adapters that can be supported in a system
- The type of adapter that can be supported on each bus
- The available memory resources

In addition, drivers should not make assumptions on a platform. Instead, they should make sure they support all the cases that are allowed by the *UEFI 2.0 Specification*. For example, memory will not always be available beneath the 4 GB boundary (some systems may not have any memory under 4 GB at all) and drivers have to be designed to be compatible with these types of system configurations. As another example, some systems do not support PC-AT* legacy hardware and drivers should not expect them to be present.

The **gBS->AllocatePool()** service does not allow the caller to specify a preferred address, so this service is always safe to use and will have no impact on platform compatibility. The **gBS->AllocatePages()** service does have a mode that allows a specific address to be specified or a range of addresses to be specified. The allocation type of **AllocateAnyPages** is safe to use and will increase platform compatibility. The allocation types of **AllocateMaxAddress** and **AllocateAddress** may reduce platform compatibility.

An UEFI driver should never directly allocate a memory buffer for DMA access. The UEFI driver cannot know enough about the system architecture to predict what system memory areas are available for DMA. Instead, an UEFI driver should use the services that are provided by the I/O protocol for the bus to allocate and free buffers available for DMA. There should also be services to initiate and complete DMA transactions. For example, the PCI Root Bridge I/O Protocol and PCI I/O Protocol both provide services for PCI DMA operations. As additional I/O bus types with DMA capabilities are introduced, new protocols that abstract the DMA services will have to be added.

5.3.2 Do Not Use Any Hard-Coded Limits

UEFI drivers should not use fixed-size arrays. Instead, memory resources should be dynamically allocated using the **gBS->AllocatePages()** and **gBS->AllocatePool()** services.

5.3.3 Do Not Make Assumptions about I/O Subsystem Configurations

UEFI drivers should assume neither a fixed nor a maximum number of controllers in a system. All UEFI drivers should be designed to manage any number of controllers even if the driver writer is convinced there will only be a fixed number of controllers. This design will maximize the compatibility of the UEFI driver, especially on multi-bus-set (ECR pending at PCI SIG) PCI systems that may contain hundreds of PCI slots. Chapter 8 introduces the private context data structure, which is a lightweight mechanism that allows an UEFI driver to be designed with no limitations on the number of controllers that the UEFI driver can manage.

5.3.4 Maximize Source Code Portability

UEFI drivers should be designed to maximize source code portability. Today, the processor targets include the following:

- IA-32
- Intel-64
- IA-64
- EBC virtual machine

It is possible to write a single driver that can be compiled for all of these processor targets. It is also possible that the list of processor targets may grow over time. As a result, assembly language is discouraged.

An UEFI driver should also never directly access any system chipset resources. Directly accessing these resources will limit the compatibility of the UEFI driver to systems only with that specific chipset. Instead, the EFI Boot Services, EFI Runtime Services, and various protocol services should be used to access the system resources that are required by an UEFI driver. Putting effort into source code portability will help maximize future platform compatibility.

5.4 UEFI Driver Model

The UEFI Driver Model goes a long way in addressing many of the potential platform differences and opens up a powerful way for drivers to interact with the user regardless of the platform specifics. The UEFI Driver Model works with existing and future bus types. UEFI drivers should be written to follow the UEFI Driver Model.

The basic structure of an UEFI driver that follows the UEFI Driver Model is defined by several basic driver protocols. Each of the protocols adds standard interface functions that are coded in the UEFI driver. All UEFI drivers needs to implement the following UEFI Driver Model–related protocols:

- **`EFI_DRIVER_BINDING_PROTOCOL`**
- **`EFI_COMPONENT_NAME_PROTOCOL`** for displaying the drivers name
- **`EFI_LOADED_IMAGE_PROTOCOL`** for unloading the driver
- **`EFI_DRIVER_CONFIGURATION_PROTOCOL`** for configuring the driver
- **`EFI_DRIVER_DIAGNOSTICS_PROTOCOL`** for diagnosing issues with the driver or hardware

For complete requirements see section 2.6.3 of UEFI 2.0 Specification.

5.5 Use the Software Abstractions

UEFI drivers shall use software abstractions that are provided and avoid the temptation of internal routines, unless the environment does not provide a suitable alternative. These software abstractions include the following:

- EFI Boot Services such as **`gBS->SetMem()`** and **`gBS->CopyMem()`**
- EFI Runtime Services such as **`gRT->GetVariable()`**
- Protocols such as the PCI I/O and Simple Text Output Protocols

This recommendation serves several purposes. By using the software abstractions provided by the platform vendor, the UEFI driver will maximize its platform compatibility. The platform vendor can also optimize the services that are provided by the platform, so the performance of the UEFI driver improves by using these services. Chapter 22 discusses the EBC porting considerations, and one of the most important considerations is the performance of an EBC driver because EBC code is interpreted. The performance of an EBC driver can be greatly improved by calling system services instead of using internal functions.

5.6 Use Polling Device Drivers

UEFI drivers are not designed to be high-performance drivers, but rather to provide basic boot support for OS loaders. For this reason, UEFI does not support an interrupt model for the device drivers. Instead, all UEFI drivers operate in a polled mode. UEFI drivers that implement blocking I/O services can simply poll the device until the request is complete. UEFI drivers that implement non-blocking I/O can create a periodic timer event to poll a device at periodic intervals. The interval should be set to the largest possible period for the UEFI driver to complete its I/O services in a reasonable period of time. The overall performance of an EFI-enabled platform will degrade if too many UEFI drivers create high-frequency periodic timer events. It is recommended that the period of a periodic timer event be at least 10 mS. In general, they should be as large as possible based upon a specific device's timing requirements, and most drivers can use events with timer periods in the range of 100 mS to several seconds. UEFI drivers should also not spend a lot of time in their event notification functions, because this blocks the normal execution mode of the system. An UEFI driver using a periodic timer event can always save some state information and wait for the next timer tick if the driver needs to wait for a device to respond. The USB bus driver is an example driver in the *EDK* that uses periodic timer events.

5.6.1 Use Events and Task Priority Levels

The TPLs provide a mechanism for code to run at a higher priority than application code. So, one can be running the UEFI Shell, and an UEFI device driver can have a timer event fire and gain control to go poll its device. The **`TPL_CALLBACK`** level is typically used for deferred software calls, and **`TPL_NOTIFY`** is typically used by device drivers. **`TPL_HIGH_LEVEL`** is typically used for locks on shared data structures.

Drivers may use events and TPLs if they perform non-blocking I/O. If they perform blocking I/O, then they will not use events. They may still use the **`gBS->RaiseTPL()`** and **`gBS->RestoreTPL()`** for critical sections.

Driver diagnostics are typically just applications. They will not normally need to use TPLs or events unless the diagnostics is testing the TPL or event mechanisms in EFI. There is one exception. If a diagnostic needs to guarantee that EFI's timer interrupt is disabled, then the diagnostic can raise the TPL to **`TPL_HIGH_LEVEL`**. If this level is required, then it should be done for the shortest possible time interval.

Please note that in the *UEFI 2.0 Specification* there are ways to have the platform firmware be put into an undefined state by misuse of the **`gBS->RaiseTPL()`** and **`gBS->RestoreTPL()`** functions.

5.7 Design to Be Re-entrant

If a system contains multiple controllers of the same type from the same vendor, it is quite possible that a single driver could install an instance of the controller's I/O protocol on a handle for each device. Each instance of the protocol would call the same driver functions, but the data would be unique to the instance (or context) of the protocol. This design concept is important for all UEFI drivers.

The practical manifestation of this requirement is that all the data that must be local to the instance (context) of the protocol must *not* be stored in global variables. Instead,

data is collected into a private context data structure and each time that an I/O protocol is installed onto a handle, a new version of the structure is allocated from memory. This concept is described in detail in chapter 8.

5.8 Avoid Function Name Collisions between Drivers

Compilers and linkers will guarantee that there are no name collisions within a driver, but the compilers and linkers cannot check for name collisions between drivers. This inability to check is a concern only when debuggers are used that can perform source-level debugging or can display function names. Section 3.12 introduced templates that help avoid function name collisions between drivers. The guidelines in the templates should be followed.

5.9 Manage Memory Ordering Issues in the DMA and Processor

Not all processors have strongly ordered memory models. This distinction means that the order in which memory transactions are presented in the source code may not be the same when the code is executed. Normally, this order is not an issue, because the processor and the compiler will guarantee that the code will execute as the developer expects. However, UEFI drivers that use DMA buffers that are simultaneously accessed by both the processor and the DMA bus master may run into issues if either the processor or the DMA bus master or both are weakly ordered. The DMA bus master must solve its own ordering issues, but the UEFI driver writer is responsible for managing the processor's ordering issues.

The *EDK* contains a macro called **`MEMORY_FENCE()`** that guarantees that all the transactions in the source code prior to the **`MEMORY_FENCE()`** macro are completed before the code after the **`MEMORY_FENCE()`** macro is executed. This macro can be used in an UEFI driver that accesses a buffer at the same time that a DMA bus master is accessing that same buffer. The UEFI driver will insert these macros at the appropriate points in the code where all the processor transactions need to be completed before the next processor transaction is performed. For example, if a buffer contains a data structure and that data structure contains a valid bit and additional data fields that describe an I/O operation that the DMA bus master needs to perform, then the UEFI driver would want to update the additional data fields before setting the valid bit. The UEFI driver would use the **`MEMORY_FENCE()`** macro just before and just after the valid bit is written to the buffer.

This issue is not present in IA-32 or EBC virtual machines because these two processor types are strongly ordered. However, Itanium-based platforms are weakly ordered, so this macro must be used for native drivers for the Itanium processor. It is recommended that these macros be used appropriately in all driver types to maximize the UEFI driver's platform compatibility.

5.10 Do Not Store UEFI drivers in Hidden Option ROM Regions

Some option ROMs may use paging or other techniques to load and execute code that was not visible to the system firmware when measuring the visible portion of the

option ROM. This technique is discouraged because it is typically the bus driver's responsibility to extract the option ROM contents when a bus is enumerated. If code is required to access hidden portions of an option ROM, then the bus driver would not have the ability to extract the additional option ROM contents. This inability means that the UEFI drivers in an option ROM must be visible without accessing a hidden portion of an option ROM. However, if there is a safe mechanism to access the hidden portions of the option ROM after the UEFI drivers have been loaded and executed, then the UEFI driver may choose to access those contents. For example, nonvolatile configuration information, utilities, or diagnostics can be stored in the hidden option ROM regions.

5.11 Store Configuration on the Same FRU as the driver

The configuration for an UEFI driver should be stored on the same Field Replaceable Unit (FRU) as the UEFI driver. If an UEFI driver is stored on the motherboard, then their configuration information can be stored in UEFI variables. If an UEFI driver is stored in an add-in card, then their configuration information should be stored in the NVRAM provided on the add-in card.

5.11.1 Benefits

This method ensures that it is possible to statically (while the FRU is being designed) determine the maximum configuration storage that is required for the FRU. In particular, if option cards were to store their configuration in UEFI variables, the amount of variable storage could not be statically calculated, because it generally is not possible to know ahead of time what set of option cards may be installed in a system. The result would be that add-in cards could not be used in otherwise functional systems due to lack of UEFI variable storage space.

Storing UEFI variables in the same FRU as the UEFI driver reduces the amount of stale data that is left in UEFI variables. If an option card was to store its data in UEFI variables and then be removed, there is no automatic cleanup mechanism to purge the UEFI variables that are associated with that card.

Storing UEFI variables in the same FRU as the UEFI driver also ensures that the configuration stays with the FRU. It enables centralized configuration of add-in cards. For example, if an IT department is configuring 50 like systems, it can configure all 50 in the same system and then disburse them to the systems, rather than configuring each system separately. Further, it can maintain preconfigured spares.

5.11.2 Update Configurations at OS Runtime Using an OS-Present Driver

If an end-user or developer wants to configure an add-in card while the OS is running, it is possible to update the configuration to the card using UEFI variables. Add-in cards are typically supplied with OS-present drivers. For most operating systems, it is actually preferable to access the card using the OS-driver and for that driver to update the card.

5.12 Do Not Use Hard-Coded Device Path Nodes

The **ACPI()** node in the EFI Device Path Protocol identifies the PCI root bridge in the ACPI namespace. The *ACPI Specification* allows _HID to describe vendor-specific capability and _CID to describe compatibility. Therefore, there is no requirement for all platforms to use the PNP0A03 identifier in the _HID to identify the PCI root bridge. The following are the only requirements for the PCI root bridge:

- The PNP0A03 identifier must appear in _HID if a vendor-specific capability does not need to be described.
- The PNP0A03 identifier must appear in _CID if _HID contains a vendor-specific identifier.

To avoid problems with platform differences, UEFI drivers should not create UEFI device paths from hard-coded information. Instead, UEFI bus drivers should append new device path nodes to the device path from the parent device handle.

5.12.1 PNPID Byte Order for EFI

The ACPI PNPID format (byte order) follows the original EISA ID format. UEFI also uses PNPID in the device path ACPI nodes. However, for a given string, ACPI and UEFI do not generate the same numbers. For example:

```
HID = "PNP0501"
ACPI = 0x0105D041
EFI = 0x050141D0
```

This difference means that operating systems that try to match the UEFI ACPI device path node to the ACPI name space must perform a translation.

5.13 Do Not Cause Errors on Shared Storage Devices

In a cluster configuration, multiple devices may be connected to a shared storage. In such configurations, the UEFI driver should not cause errors that can be seen by the other devices that are connected to storage.

On a boot or reboot, there shall be no writes to shared storage without user acknowledgement. Any writes to shared storage by an UEFI driver may corrupt shared storage as viewed by another system. As a result, all outstanding I/O in the controller's buffers shall be cleared, and any internal caches shall also be cleared. Any I/O operations that occur after a reboot may corrupt shared storage.

There must not be an excessive number of bus or device resets. Device resets have an impact on shared storage as viewed by other systems. For a single reset, this impact is negligible. Larger numbers of resets may be seen as a device failure by another system.

Disk signatures must not be changed without warning the user. If there is an impact to the user, then that impact should be displayed along with the warning. Clusters may make an assumption about disk signatures on shared storage.

The discovery process must not impact other systems accessing the storage. A long discovery process may "hold" drives and look like a failure of shared storage.

5.14 Convert Bus Walks

In UEFI systems, the UEFI bus drivers find, enumerate, and expose devices using the bus's standard I/O protocol. For example, the PCI bus driver exposes devices on the PCI bus that have device handles that support the PCI I/O Protocol. The USB bus driver exposes USB devices that have a handle that supports the USB I/O Protocol. The list of device handles that support a specific bus I/O protocol can be discovered using the **`gBS->LocateHandleBuffer()`** service. Not only is this method faster than a classical bus walk, but it will work even when a system contains multiple PCI buses.

5.14.1 Example

Example 65 shows an example of converting a bus walk on the PCI bus. In this example, use EFI Boot Services to obtain all the handles that support the PCI I/O Protocol and then examine the configuration space for the vendor ID (VID) and device ID (DID). Note that the code below could be used in a flash utility to discover all the cards in the system.

```
EFI_STATUS              Status;
UINTN                   HandleCount;
EFI_HANDLE              *HandleBuffer;
UINTN                   Index;
EFI_PCI_IO_PROTOCOL     *PciIo;
PCI_TYPE00              Pci;

//
// Retrieve the list of handles that support the PCI I/O Protocol from
// the handle database. The number of handles that support the PCI I/O
// Protocol is returned in HandleCount, and the array of handle values is
// returned in HandleBuffer.
//
Status = gBS->LocateHandleBuffer (
                ByProtocol,
                &gEfiPciIoProtocolGuid,
                NULL,
                &HandleCount,
                &HandleBuffer
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Loop through all the handles that support the PCI I/O Protocol, and
// retrieve the instance of the PCI I/O Protocol.
//
for (Index = 0; Index < HandleCount; Index++) {
  Status = gBS->OpenProtocol (
                  HandleBuffer[Index],
                  &gEfiPciIoProtocolGuid,
                  (VOID **)&PciIo,
                  ImageHandle,
                  NULL,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
  if (EFI_ERROR (Status)) {
    continue;
  }

  DEBUG ((D_INIT, "DrvUtilApp: Call Pci.Read \n"));
  Status = PciIo->Pci.Read (
                        PciIo,
                        EfiPciIoWidthUint8,
                        0,
                        sizeof (Pci),
                        &Pci
                        );

  if (EFI_ERROR (Status)) {
    Status = EFI_UNSUPPORTED;
    DEBUG ((D_INIT, "DrvUtilApp: Return Value = %r\n",Status));
  } else {

    //
    // Check for desired attributes:
    //  Pci.Hdr.VendorId
    //  Pci.Hdr.DeviceId
    //

    if ((Pci.Hdr.VendorId != XYZ_VENDOR_ID) ||
        (Pci.Hdr.DeviceId != XYZ_DEVICE_ID)) {
      DEBUG((D_INIT, "DrvUtilApp: This DH not ours: %d\n",
HandleBuffer[Index]));
    } else {
      DEBUG((D_INIT, "DrvUtilApp: FOUND! %d\n", HandleBuffer[Index]));
    }
```

```
  }
}
}
//
// Free the array of handles that was allocated by gBS-
>LocateHandleBuffer()
//
gBS->FreePool (HandleBuffer);
```

Example 65. PCI Bus Walk Example

5.15 Do Not Have Any Console Display or Hot Keys

PC BIOS legacy option ROMs would typically display banners. The header displays the driver name and version information. Some BIOS drivers inform the user of a "hot key" that would be checked during the boot sequence that would allow customers to enter into the driver's configuration options. This type of interaction created several problems:

- Displaying the banner and waiting long enough to detect a key press increases boot time. On systems where there could be hundreds of PCI cards, this time can add up. Even the time to print a header can take too long, because some systems may require the console text to flow out to a serial port that could be configured at 9600 baud.
- The user is rarely ready to interact with the boot driver at just the right moment and often has to reboot the system multiple times to read the display banner or press the key at the right time to enter the driver configuration utility. On larger systems, the system boot can take many minutes.
- The amount of text "spew" can be confusing to a user watching the boot sequence. Each driver can have its own unique way of presenting the information. When many cards are all displaying their own unique header information to the screen at once, there is a significant risk of confusing the user with too much information that is only informational and not significant to the health of the system. Also, the text can unnecessarily fill console logs and potentially break scripts when a new card is added to a system that was monitoring the console text.
- On fast systems, the header text is displayed but erased by a later driver so quickly that the user cannot read it.
- In EFI, the driver may be initialized before the console output and standard error devices. In such cases, the data will not be visible or will crash the system. When UEFI option ROM drivers are loaded, the console has not been connected because the console may in fact be controlled by one of the UEFI option ROM drivers. In such cases, printing may in fact call a **NULL** function and crash the system.

For these reasons, the UEFI Driver Model requires that no console I/O operations take place in the UEFI driver Binding Protocol functions. A reasonable exception to this rule is to use the **DEBUG()** macro to display progress information during driver development and debug. Using the **DEBUG()** macro allows the code for displaying the data to be easily removed for a production build of the driver.

Use of the **DEBUG()** macro should be limited to "debug releases" of a driver. This strategy will typically work if the driver is loaded after the UEFI console is connected. However, some firmware implementations may load the option ROM drivers before the UEFI console is connected (because console drivers may live in option ROMs). In such

cases, the *ConOut* and *StdErr* fields of the EFI system table may be **NULL**, and printing will crash the system. The **DEBUG()** macro should check to see if the field is **NULL** before using those services.

Chapter 10 of the *UEFI 2.0 Specification* provides several other protocols to assist in interacting with the user— **EFI_DRIVER_CONFIGURATION_PROTOCOL** and **EFI_DRIVER_DIAGNOSTICS_PROTOCOL**.

5.16 Offer Alternatives to Function Key

The UEFI console may be connected through a serial port. In such cases, it is sensitive to the correct terminal emulator configuration. If the user has not correctly configured the terminal emulator to match the terminal settings in UEFI (PC ANSI, VT100, VT100+, or VT-UTF8), they may not be able to correctly use some of the keys (function keys, arrow keys, page up/down, insert/delete, and backspace) or be able to display colors and see the correct cursor positioning. To better support users, it is recommended that UEFI configuration protocols and UEFI applications create user interfaces that are not solely dependent on these keys but instead offer alternatives for these keys. Also, it is important to note that the Simple Input Protocol does not support the CTRL or ALT keys because these keys are not available with remote terminals such as terminal emulators and telnet. Table 22 below shows one possible set of alternate key sequences for function keys, arrow keys, page up/down keys, and the insert/delete keys. Each configuration protocol and application will have to decide if alternate key sequences are supported and which alternate mappings should be used. Table 22 lists the EFI Scan Code from the Simple Input Protocol and the alternate key sequence that can be used to produce that scan code. Most of these key sequences are directly supported in the *EDK*, which means that the developer does not need to do anything special to support these key sequences on a remote terminal. The ones labeled as "No" are not directly supported in the *EDK*, so those key sequences will have to be parsed to be interpreted by the configuration protocol or application.

Table 22. Alternate Key Sequences for Remote Terminals

EFI Scan Code	Key Sequence	Supported in EDK?
SCAN_NULL		
SCAN_UP	'^'	No
SCAN_DOWN	'v' or 'V'	No
SCAN_RIGHT	'>'	No
SCAN_LEFT	'<'	No
SCAN_HOME	ESC h	Yes
SCAN_END	ESC k	Yes
SCAN_INSERT	ESC +	Yes
SCAN_DELETE	ESC -	Yes
SCAN_PAGE_UP	ESC ?	Yes
SCAN_PAGE_DOWN	ESC /	Yes
SCAN_F1	ESC 1	Yes

SCAN_F2	ESC 2	Yes
SCAN_F3	ESC 3	Yes
SCAN_F4	ESC 4	Yes
SCAN_F5	ESC 5	Yes
SCAN_F6	ESC 6	Yes
SCAN_F7	ESC 7	Yes
SCAN_F8	ESC 8	Yes
SCAN_F9	ESC 9	Yes
SCAN_F10	ESC 0	Yes
ESC	ESC	Yes

5.17 Do Not Assume UEFI Firmware Will Execute All Drivers

Typically, the same vendor that produces an UEFI driver will also have to produce an OS-present driver for all the operating systems that the vendor chooses to support. Because UEFI provides a mechanism to reduce the boot time by running the minimum set of drivers that are required to connect the console and boot devices, not all UEFI drivers may be executed on every boot. For example, the system may have three SCSI cards but only needs to install the driver on one SCSI bus to boot the OS.

This minimum set of drivers means that the OS-present driver may be handed a controller that may be in several different states. It may still be in the power-on reset state, it may have been managed by an UEFI driver for a short period of time and released, and it may have been managed by an UEFI driver right up to the point in time where firmware hands control of the platform to the operating system.

The OS-present driver must accept controllers in all of these states. This acceptance requires the OS-present driver to make very few assumptions about the state of the controller it manages.

OS drivers shall not make assumptions that the UEFI driver has initialized or configured the device in any way. Also, I/O hot-plug does not involve UEFI driver execution, so the OS driver must be able to initialize and operate the driver without UEFI support.

6 Classes of UEFI drivers

The different classes of UEFI drivers are introduced in chapter 2. These driver classes will be discussed throughout this document, but an emphasis is placed on drivers that follow the UEFI Driver Model because these drivers are the most commonly implemented. The driver classes that follow the UEFI Driver Model include the following:

- Device drivers
- Bus drivers
- Hybrid drivers

There are actually several subtypes and optional features for these classes of drivers. This chapter introduces the subtypes and optional features of drivers that follow the UEFI Driver Model. Understanding the different classes of UEFI drivers will help driver writers identify the class of driver to implement and the algorithms that are used in their implementation. The less common service drivers, root bridge drivers, and initializing drivers are also discussed.

The chapters that follow describe a driver's entry point in detail, including the various optional features that may be enabled in the entry point. The driver entry point chapter is followed by a chapter on using object-oriented programming techniques to help design good data structures in drivers that follow the UEFI Driver Model. This chapter is followed by a chapter on the Driver Binding Protocol and chapters on the optional protocols such as Component Name, Driver Configuration, Driver Diagnostics, Bus Specific Driver Override, and Platform Driver Override.

6.1 Device Drivers

All device drivers that follow the UEFI Driver Model share a set of common characteristics. The following two sections describe the required and optional features for device drivers. These sections are followed by a detailed description of device drivers that produce a single instance of the Driver Binding Protocol and device drivers that produce multiple instances of the Driver Binding Protocol.

6.1.1 Required Device Driver Features

Device drivers are required to implement the following features:

- Installs one or more instances of the **EFI_DRIVER_BINDING_PROTOCOL** in the driver's entry point.
- Manages one or more controller handles. Even if a driver writer is convinced that the driver will manage only a single controller, the driver should be designed to manage multiple controllers. The overhead for this functionality is low, and it will make the driver more portable.
- Does not produce any child handles. This feature is the main distinction between device drivers and bus/hybrid drivers.
- Ignores the *RemainingDevicePath* parameter that is passed into the **Supported()** and **Start()** services of **EFI_DRIVER_BINDING_PROTOCOL**.
- Consumes one or more I/O-related protocols from the controller handle.
- Produces one or more I/O-related protocols on the same controller handle.

6.1.2 Runtime Device Driver Features

- Creates a Set Virtual Address Map event in the driver's entry point. This feature is required only for a device driver that is an UEFI runtime driver with pointers in its private data structure.
- Creates an Exit Boot Services event in the driver's entry point. This feature is required if the driver needs to act when the OS loader is switching from EFI Boot Services time to EFI Runtime Services time.

6.1.3 Optional Device Driver Features

Each of the following features is optional, but strongly suggested to be implemented by each driver, but is not required:

- Creates an Exit Boot Services event in the driver's entry point. This feature is required only if the driver is required to place the devices it manages in a specific state just before control is handed to an operating system.
- Installs one or more instances of the **EFI_COMPONENT_NAME_PROTOCOL** in the driver's entry point. Implementing this feature is strongly recommended. It allows a driver to provide human-readable names for the name of the driver and the controllers that the driver is managing. Some platform design guides such as *DIG64* require this feature.
- Installs one or more instances of the **EFI_DRIVER_CONFIGURATION_PROTOCOL** in the driver's entry point. If a driver has any configurable options, then this protocol is required. Some platform design guides such as *DIG64* require this feature.
- Installs one or more instances of the **EFI_DRIVER_DIAGNOSTICS_PROTOCOL** in the driver's entry point. If a driver wishes to provide diagnostics for the controllers that the driver manages, then this protocol is required. This protocol is the only mechanism that is available to a driver when the driver wants to

alert the user of a problem that was detected with a controller. Some platform design guides such as *DIG64* require this feature.

- Provides an **EFI_LOADED_IMAGE_PROTOCOL.Unload()** service, so the driver can be dynamically unloaded. It is recommended that this feature be implemented during driver development, driver debug, and system integration. It is strongly recommended that this service remain in drivers for add-in adapters to help debug interaction issues during system integration.

6.1.4 Device Drivers with One Driver Binding Protocol

Most device drivers produce a single instance of the **EFI_DRIVER_BINDING_PROTOCOL**. These drivers are the simplest that follow the UEFI Driver Model, and all other driver types have their roots in this type of device driver.

A device driver is loaded into memory with the **gBS->LoadImage()** Boot Service and invoked with the **gBS->StartImage()** Boot Service. The **gBS->LoadImage()** service automatically creates an image handle and installs the **EFI_LOADED_IMAGE_PROTOCOL** onto the image handle. The **EFI_LOADED_IMAGE_PROTOCOL** describes the location where the device driver was loaded and the location in system memory where the device driver was placed. The **Unload()** service of the **EFI_LOADED_IMAGE_PROTOCOL** is initialized to **NULL** by **gBS->LoadImage()**. This setting means that by default the driver is not unloadable. The **gBS->StartImage()** service transfers control to the driver's entry point as described in the PE/COFF header of the driver image. The driver entry point is responsible for installing the **EFI_DRIVER_BINDING_PROTOCOL** onto the driver's image handle. Figure 9 below shows the state of the system before a device driver is loaded, just before it is started, and after the driver's entry point has been executed.

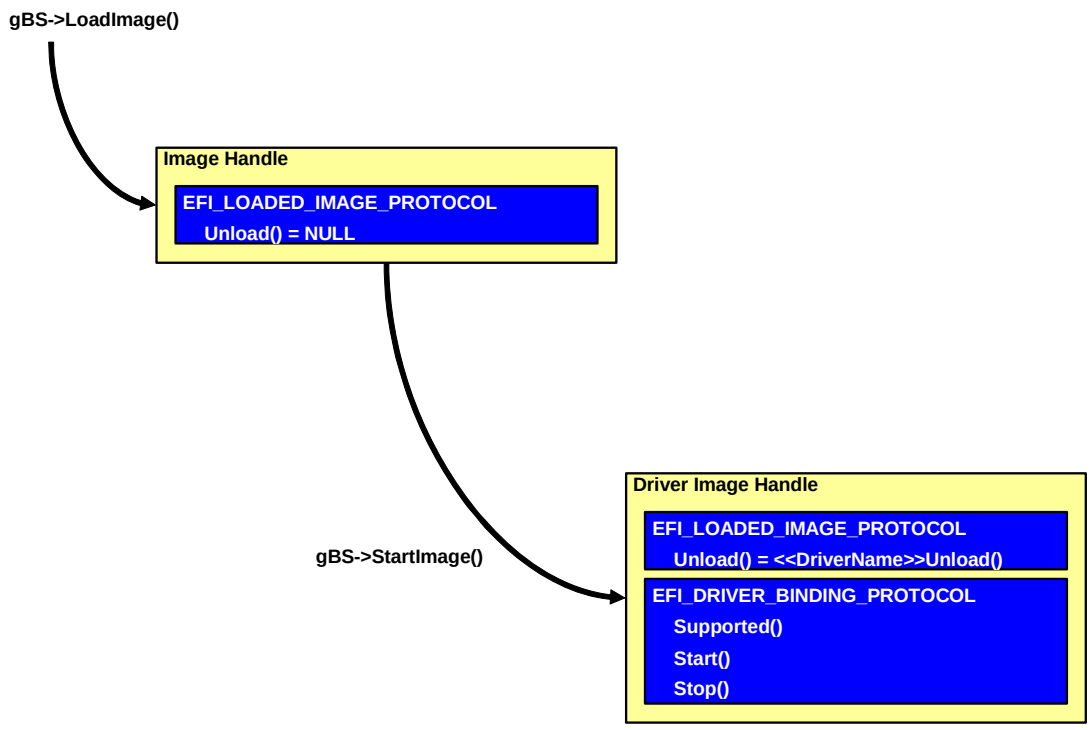

Figure 9. Device Driver with Single Driver Binding Protocol

Figure 10 below is the same as the figure above, except this device driver has also implemented all the optional features. This difference means the following:

- Additional protocols are installed onto the driver's image handle.
- An **`Unload()`** service is registered in the **`EFI_LOADED_IMAGE_PROTOCOL`**.
- An Exit Boot Services event and Set Virtual Address Map event have been created.

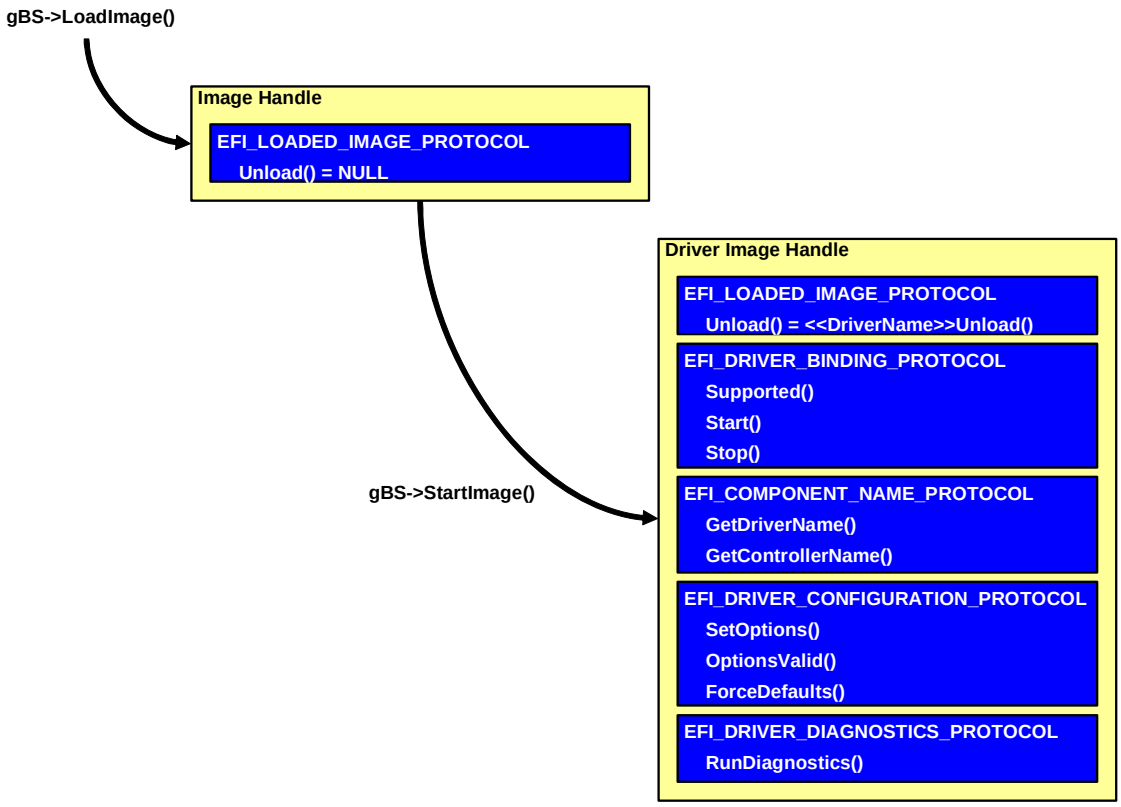

Figure 10. Device Driver with Optional Features

Section 6.1.4.1 below lists some of the device drivers from the *EDK* that produce a single instance of the **`EFI_DRIVER_BINDING_PROTOCOL`**.

6.1.4.1 Sample Device Drivers with One Driver Binding Protocol in the EDK

- AtapiPassThru
- CirrusLogic5430
- DiskIo
- FileSystem\Fat
- IsaFloppy
- PcatIsaAcpi
- PcatIsaAcpiBios
- PciVgaMiniPort
- Ps2Keyboard

- Ps2Mouse
- PxeBc
- PxeDhcp4
- ScsiDisk
- Snp32_64
- Usb\Uhci
- Usb\UsbBot
- Usb\UsbCbi
- Usb\UsbKb
- Usb\UsbMassStorage
- Usb\UsbMouse
- VgaClass
- WinNtThunk\BlockIo
- WinNtThunk\Console
- WinNtThunk\SimpleFileSystem
- BiosInt\BiosKeyboard
- BiosInt\BiosSnp16
- BiosInt\BiosVideo

6.1.5 Device Drivers with Multiple Driver Binding Protocols

A more complex device driver is one that produces more than one instance of the **EFI_DRIVER_BINDING_PROTOCOL**. The first instance of the **EFI_DRIVER_BINDING_PROTOCOL** is installed onto the driver's image handle, and the additional instances of the **EFI_DRIVER_BINDING_PROTOCOL** are installed onto newly created driver binding handles.

Figure 11 below shows the state of the handle database before a driver is loaded, before it is started, and after its driver entry point has been executed. This specific driver produces three instances of the **EFI_DRIVER_BINDING_PROTOCOL**.

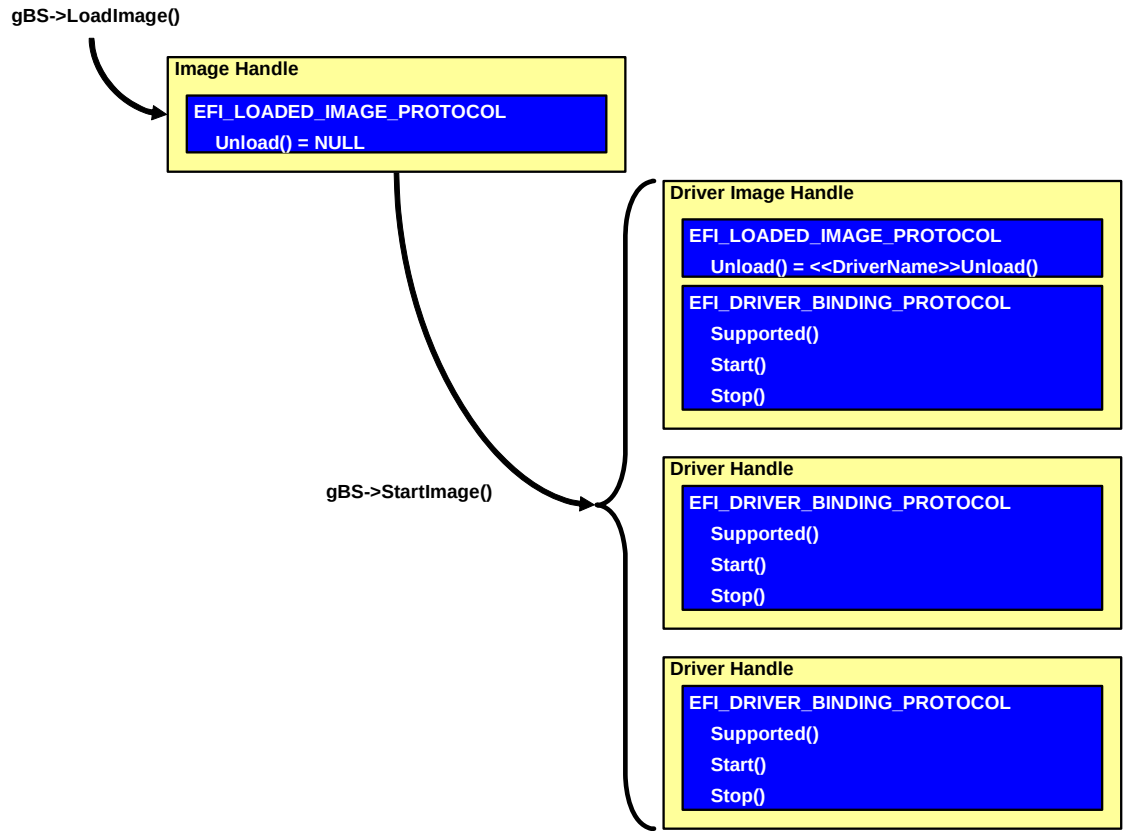

Figure 11. Device Driver with Multiple Driver Binding Protocols

Any device driver that produces multiple instances of the **EFI_DRIVER_BINDING_PROTOCOL** can be broken up into multiple drivers, so each driver produces a single instance of the **EFI_DRIVER_BINDING_PROTOCOL**. However, there are a few advantages for a driver to produce multiple instances of the **EFI_DRIVER_BINDING_PROTOCOL**. The first is that it may reduce the overall size of the drivers. If two related drivers are combined and those two drivers can share internal functions, the executable image size of the single driver may be smaller than the sum of the two individual drivers. The other reason to combine drivers is to help manage platform features. A single platform features may require several drivers. If the drivers are separated, then multiple drivers have to be added or removed to add or remove that single feature.

There is one device driver in the *EDK* that produces multiple instances of the **EFI_DRIVER_BINDING_PROTOCOL**, and it is the console platform driver in the **\Sample\Platform\Generic\Dxe\ConPlatform** subdirectory. This driver implements the platform policy for managing multiple console input and console out devices. It produces one **EFI_DRIVER_BINDING_PROTOCOL** for the console output devices, and another **EFI_DRIVER_BINDING_PROTOCOL** for the console input devices. The management of console devices needs to be centralized, so it makes sense to combine

these two functions into a single driver so the platform vendor needs to update only one driver to adjust the platform policy for managing console devices.

6.1.6 Device Driver Protocol Management

Device drivers consume one or more I/O-related protocols and use the services of those protocols to produce one or more I/O-related protocols. The **Supported()** and **Start()** functions of the **EFI_DRIVER_BINDING_PROTOCOL** are responsible for opening the I/O-related protocols that are being consumed using the EFI Boot Service **gBS->OpenProtocol()**. The **Stop()** function is responsible for closing the consumed I/O-related protocols using **gBS->CloseProtocol()**. A protocol can be opened in several different modes, but the most common is **BY_DRIVER**. When a protocol is opened **BY_DRIVER**, a test is made to see if that protocol is already being consumed by any other drivers. The open operation will succeed only if the protocol is not being consumed by any other drivers. This use of the **gBS->OpenProtocol()** service is how resource conflicts are avoided in the UEFI Driver Model. However, it requires that every driver present in the system to follow the driver interoperability rules for all resource conflicts to be avoided.

Figure 12 below shows the image handle for a device driver as **gBS->LoadImage()** and **gBS->StartImage()** are called. In addition, it shows the states of three different controller handles as the **EFI_DRIVER_BINDING_PROTOCOL** services **Supported()**, **Start()**, and **Stop()** are called. *Controller Handle 1* and *Controller Handle 3* pass the **Supported()** test, so the **Start()** function can be called. In this case, the **Supported()** service tests to see if the controller handle supports Protocol A. **Start()** is then called for *Controller Handle 1* and *Controller Handle 3*. In the **Start()** function, *Protocol A* is opened **BY_DRIVER**, and *Protocol B* is installed onto the same controller handle. The implementation of *Protocol B* will use the services of *Protocol A* to produce the services of *Protocol B*. All drivers that follow the UEFI Driver Model must support the **Stop()** service. The **Stop()** service must put the handles back into the same state they were in before **Start()** was called, so the **Stop()** service uninstalls *Protocol B* and closes *Protocol A*.

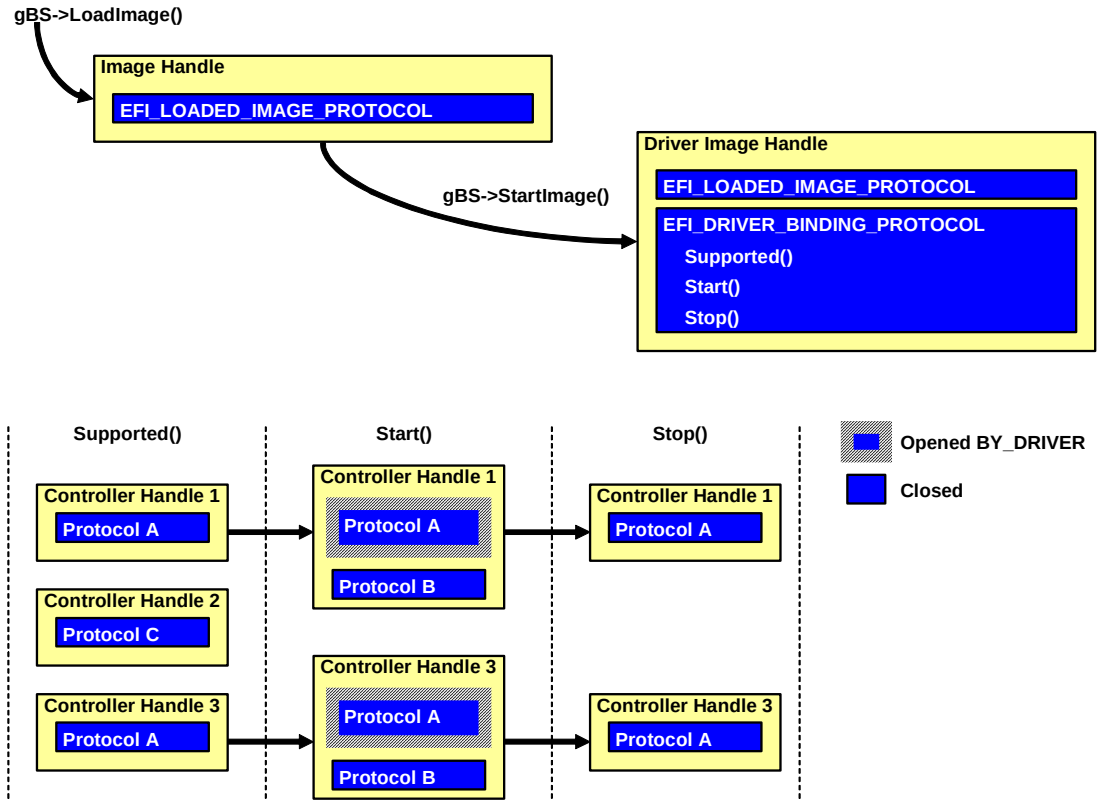

Figure 12. Device Driver Protocol Management

Figure 13 below shows a more complex device driver that requires *Protocol A* and *Protocol B* to produce *Protocol C*. Notice that the controller handles that support neither *Protocol A* nor *Protocol B*, only *Protocol A*, or only *Protocol B* do not pass the **`Supported()`** test. Also notice that *Controller Handle 6* already has *Protocol A* opened **`BY_DRIVER`**, so this device driver that requires both Protocol A and Protocol B will not pass the **`Supported()`** test either. This example highlights some of the flexibility of the UEFI Driver Model. Because the **`Supported()`** and **`Start()`** services are functions, a driver writer can implement simple or complex algorithms to test if the driver supports a specific controller handle.

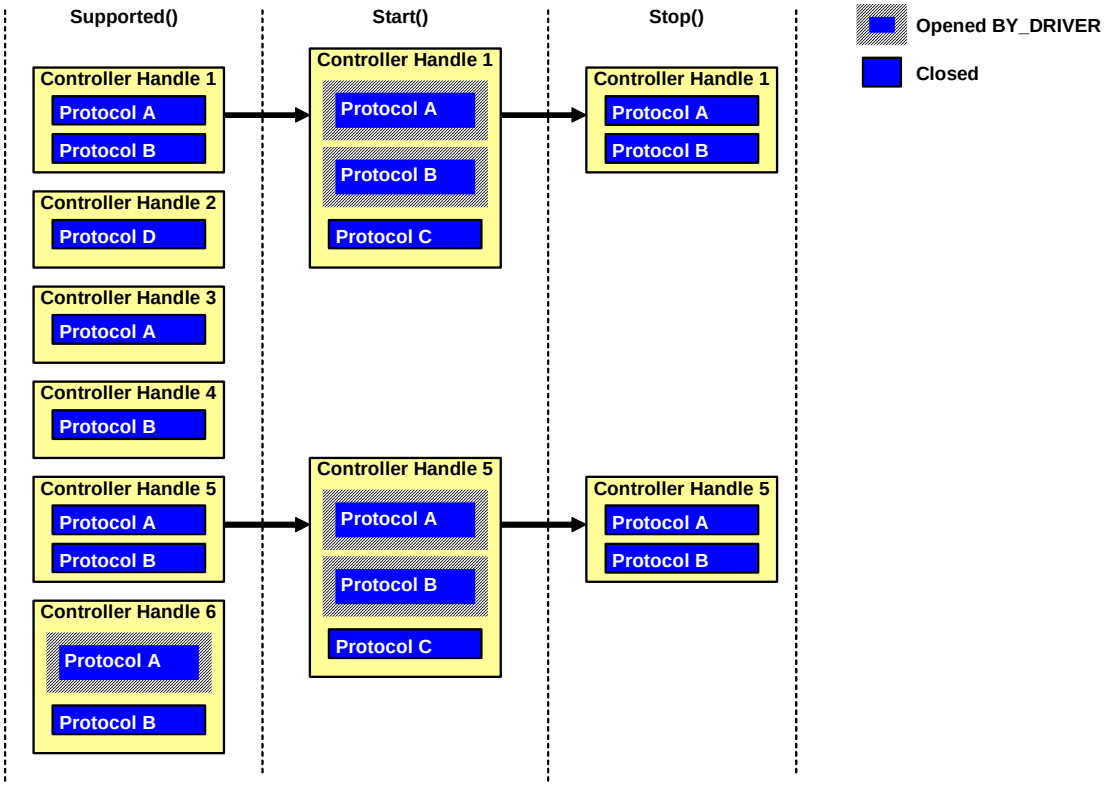

Figure 13. Complex Device Driver Protocol Management

The best way to design the algorithm for the opening protocols is to write a Boolean expression for the protocols that a device driver consumes. Then, expand this Boolean expression into the sum of products form. Each product in the expanded expression requires its own **EFI_DRIVER_BINDING_PROTOCOL**. This scenario is another way that a device driver may be required to produce multiple instances of the **EFI_DRIVER_BINDING_PROTOCOL**. The **Supported()** service for each **EFI_DRIVER_BINDING_PROTOCOL** will attempt to open each protocol in a product term. If any of those open operations fail, then **Supported()** fails. If all the opens succeed, then the **Supported()** test passes. The **Start()** function should open each protocol in the product term, and the **Stop()** function should close each protocol in the product term.

For example, the two examples above would have the following Boolean expressions:

(Protocol A)

(Protocol A AND Protocol B)

These two expressions have only one product term, so only one **EFI_DRIVER_BINDING_PROTOCOL** is required. A more complex expression would be as follows:

(Protocol A AND (Protocol B OR Protocol C))

If this Boolean expression is expanded into a sum of product form, it would yield the following:

((Protocol A AND Protocol C) OR (Protocol B AND Protocol C))

This expression would require a driver with two instances of the **EFI_DRIVER_BINDING_PROTOCOL**. One would test for *Protocol A* and *Protocol C*, and the other would test for *Protocol B* and *Protocol C*.

6.2 Bus Drivers

All bus drivers that follow the UEFI Driver Model share a set of common characteristics. The following two sections describe the required and optional features for bus drivers. These sections are followed by a detailed description of bus drivers that do the following:

6.2.3

Produce a single instance of the Driver Binding Protocol

6.2.4

Produce multiple instances of the Driver Binding Protocol

6.2.7

Produce all of their child devices in their **Start()** function

6.2.6

Are able to produce a single child device in their **Start()** function

6.2.8.1

Produce at most one child device from their **Start()** function

6.2.9

Bus drivers that do not produce any child devices in their **Start()** function

6.2.10

Produce child devices with multiple parent devices

6.2.1 Required Bus Driver Features

Bus drivers are required to implement the following features:

- Installs one or more instances of the **EFI_DRIVER_BINDING_PROTOCOL** in the driver's entry point.
- Manages one or more controller handles. Even if a driver writer is convinced that the driver will manage only a single bus controller, the driver should be

designed to manage multiple bus controllers. The overhead for this functionality is low, and it will make the driver more portable.

- Produces any child handles. This feature is the key distinction between device drivers and bus drivers.
- Consumes one or more I/O-related protocols from a controller handle.
- Produces one or more I/O-related protocols on each child handle.

6.2.2 Optional Bus Driver Features

Bus drivers can optionally implement the following features:

Installs one or more instances of the `EFI_COMPONENT_NAME_PROTOCOL` in the driver's entry point.

The implementation of this feature is strongly recommended. It allows a driver to provide human-readable names for the driver and the controllers that the driver is managing. Bus drivers should also provide names or the child handles created by the bus driver. Some platform design guides such as *DIG64* require this feature.

Installs one or more instances of the `EFI_DRIVER_CONFIGURATION_PROTOCOL` in the driver's entry point.

If a driver has any configurable options for the controller or the children, then this protocol is required. Some platform design guides such as *DIG64* require this feature.

Installs one or more instances of the `EFI_DRIVER_DIAGNOSTICS_PROTOCOL` in the driver's entry point.

If a driver wishes to provide diagnostics for the controllers for the children that the driver manages, then this protocol is required. This protocol is the only mechanism that is available to a driver when the driver wants to alert the user of a problem that was detected with a controller. Some platform design guides such as *DIG64* require this feature.

Provides an `EFI_LOADED_IMAGE_PROTOCOL.Unload()` service, so the driver can be dynamically unloaded.

It is recommended that this feature be implemented during driver development, driver debug, and system integration. It is strongly recommended that this service remain in drivers for add-in adapter to help debug interaction issues during system integration.

Parses the *`RemainingDevicePath`* parameter that is passed into the `Supported()` and `Start()` services of the `EFI_DRIVER_BINDING_PROTOCOL`.

Installs an `EFI_DEVICE_PATH_PROTOCOL` on each child handle that is created.

This feature is required only if the child handle represents a physical device. If child handle represents a virtual device, then an **`EFI_DEVICE_PATH_PROTOCOL`** is not required.

Creates an Exit Boot Services event in the driver's entry point.

This feature is required only if the driver is required to place the devices it manages in a specific state just before control is handed to an operating system.

Creates a Set Virtual Address Map event in the driver's entry point.

This feature is required only for a device driver that is also an EFI runtime driver.

6.2.3 Bus Drivers with One Driver Binding Protocol

The driver entry point of a bus driver is very similar to the driver entry point of a device driver. The discussion in section 6.1.4 applies equally well to both bus drivers and device drivers. The differences between bus drivers and device drivers are exposed in the implementations of the **EFI_DRIVER_BINDING_PROTOCOL**. The following sections describe the behaviors of the **Start()** function of the **EFI_DRIVER_BINDING_PROTOCOL** for each type of bus driver.

Section 6.2.3.1 below lists the bus drivers from the *EDK* that produce a single instance of the **EFI_DRIVER_BINDING_PROTOCOL**.

6.2.3.1 Sample Bus Drivers with One Driver Binding Protocol in the EDK

- DebugPort
- Partition
- PciBus
- ScsiBus
- Undi
- WinNtSerialIo
- BiosSnp16

6.2.4 Bus Drivers with Multiple Driver Binding Protocols

The driver entry point of a bus driver is very similar to the driver entry point of a device driver. The discussion in section 6.1.5 applies equally well to both bus drivers and device drivers. The differences between bus drivers and device drivers are exposed in the implementations of the **EFI_DRIVER_BINDING_PROTOCOL**. The following sections describe the behaviors of the **Start()** function of the **EFI_DRIVER_BINDING_PROTOCOL** for each type of bus driver.

There is only one bus driver in the *EDK* that produces multiple instances of the **EFI_DRIVER_BINDING_PROTOCOL**, and this driver is the console splitter driver. This driver multiplexes multiple console output and console input devices into a single virtual console device. It produces instances of the **EFI_DRIVER_BINDING_PROTOCOL** for the following:

- Console output devices
- Standard error device
- Console input device
- Simple pointer devices

This driver is an example of a single feature that can be added or removed from a platform by adding or removing a single component. It could have been implemented as four different drivers, but there were many common functions between the drivers, so it also saved code space to combine these four functions.

6.2.5 Bus Driver Protocol and Child Management

The management of I/O-related protocols by a bus driver is very similar to the management of I/O-related protocol for device drivers that is described in section 6.1.6. A bus driver opens one or more I/O-related protocols on the controller handle for the bus controller, and it creates one or more child handles and installs one or more I/O-related protocols. If the child handle represents a physical device, then a Device Path Protocol must also be installed onto the child handle. The child handle is also required to open the parent I/O protocol with an attribute of **BY_CHILD_CONTROLLER**.

Some types of bus drivers can produce a single child handle each time **Start()** is called, but only if the *RemainingDevicePath* passed into **Start()** represents a valid child device. This distinction means that it may take multiple calls to **Start()** to produce all the child handles. If *RemainingDevicePath* is **NULL**, than all of the remaining child handles will be created at once. When a bus driver opens an I/O-related protocol on the controller handle, it will typically use an open mode of **BY_DRIVER**. However, depending on the type of bus driver, a return code of **EFI_ALREADY_STARTED** from **gBS->OpenProtocol()**may be acceptable. If a device driver gets this return code, then the device driver should not manage the controller handle. If a bus driver gets this return code, then it means that the bus driver has already connected to the controller handle at some point in the past.

Figure 14 below shows a simple bus driver that consumes *Protocol A* from a bus controller handle and creates N child handles with a *Device Path Protocol* and *Protocol B*. The **Stop()** function is responsible for destroying the child handles by removing *Protocol B* and the *Device Path Protocol*. Protocol A is first opened **BY_DRIVER** so *Protocol A* cannot be requested by any other drivers. Then, as each child handle is created, the child handle opens *Protocol A* **BY_CHILD_CONTROLLER**. Using this attribute records the parent-child relationship in the handle database, so this information can be extracted if needed. The parent-child links are used by **gBS->DisconnectController()** when a request is made to stop a bus controller.

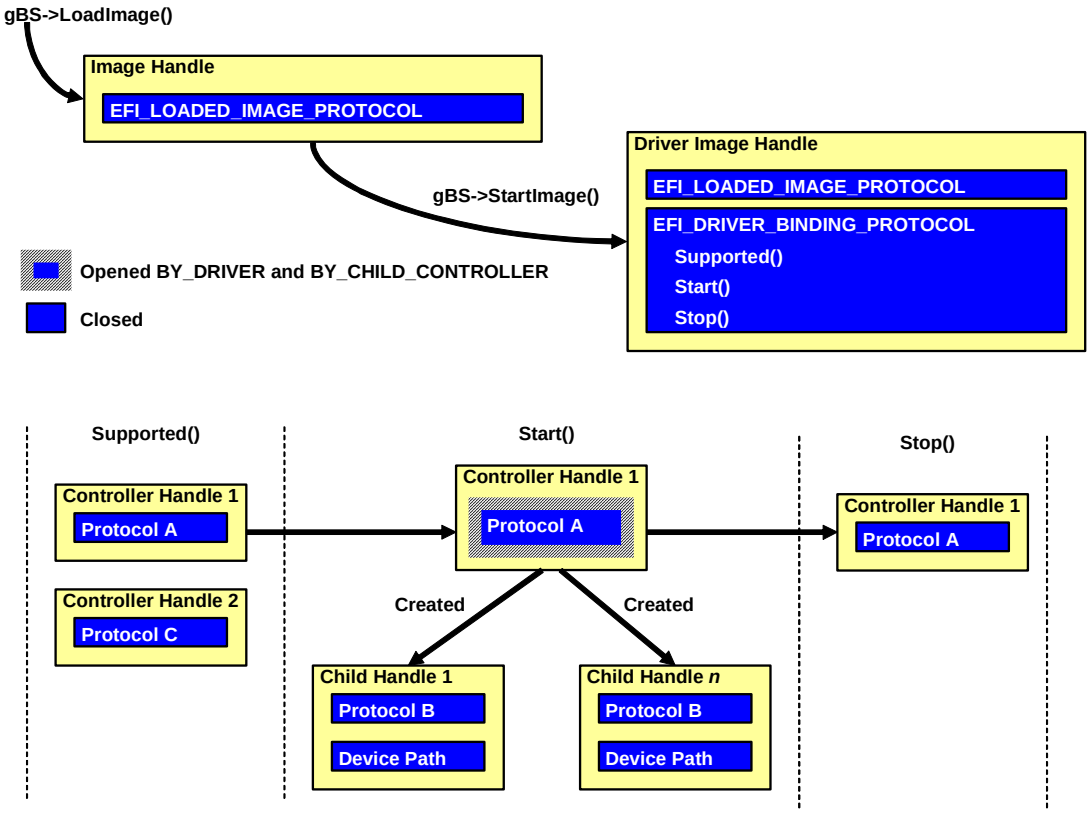

Figure 14. Bus Driver Protocol Management

The following sections describe the subtle differences in the child handle creation for each of the bus driver types.

6.2.6 Bus Drivers That Produce One Child in Start()

If the *RemainingDevicePath* parameter passed into **Supported()** and **Start()** is **NULL**, then the bus driver must produce child handles for all the children. If *RemainingDevicePath* is not **NULL**, then the bus driver should parse *RemainingDevicePath* and attempt to produce only the one child device that is described by *RemainingDevicePath*. If the driver does not recognize the device path node in *RemainingDevicePath*, or if the device that is described by the device path node does not match any of the children that are currently attached to the bus controller, then the **Supported()** and **Start()** services should fail. If the *RemainingDevicePath* is recognized and the device path node does match a child device that is attached to the bus controller, then a child handle should be created for that one child device. This step does not make sense for all bus types, because some bus types require the entire bus to be enumerated to produce even a single child. In these cases, the *RemainingDevicePath* should be ignored.

If a bus type has the ability to produce a child handle without enumerating the entire bus, then this ability should be implemented. Implementing this feature will provide faster boot times and is one of the major advantages of the UEFI Driver Model. The UEFI boot manager may pass the *RemainingDevicePath* of the console device and boot devices to **gBS->ConnectController()**, and **gBS->ConnectController()** will pass this same *RemainingDevicePath* into the **Supported()** and **Start()** services of the **EFI_DRIVER_BINDING_PROTOCOL**. This design allows the minimum number of drivers to be started to boot an operating system. This process can be repeated, so one additional child handle can be produced in each call to **Start()**. Also, a few child handles can be created from the first few calls to **Start()** and then a *RemainingDevicePath* of **NULL** may be passed in, which would required the rest of the child handle to be produced. For example, most SCSI buses do not need to be scanned to create a handle for a SCSI device whose SCSI PUN and SCSI LUN is known ahead of time. By starting only the single hard disk on a SCSI channel that is required to boot an operating system, the scanning of all the other SCSI devices can be eliminated.

Section 6.2.6.1 below lists the bus drivers from the *EDK* that can produce a single child handle in the **Start()** service of the **EFI_DRIVER_BINDING_PROTOCOL**.

6.2.6.1 Sample Bus Drivers That Produce One Child in Start()

- Ide
- PciBus
- ScsiBus
- WinNtThunk\WinNtBusDriver

6.2.7 Bus Drivers That Produce All Children in Start()

If a bus driver is always required to enumerate all of its child devices, then the *RemainingDevicePath* parameter should be ignored in the **Supported()** and **Start()** services of the **EFI_DRIVER_BINDING_PROTOCOL**. All of the child handles should be produced in the first call to **Start()**.

6.2.7.1 below lists the bus drivers from the *EDK* that produce all of the child handles in the first call to the **Start()** service of the **EFI_DRIVER_BINDING_PROTOCOL**.

6.2.7.1 Sample Bus Drivers That Produce All Children in Start()

- Console\ConSplitter
- IsaBus
- Partition

6.2.8 Bus Drivers That Produce at Most One Child in Start()

Some bus drivers are for bus controllers that have only a single port, so they have at most one child handle. If *RemainingDevicePath* is **NULL**, then that one child handle should be produced. If *RemainingDevicePath* is not **NULL**, then the *RemainingDevicePath* should be parsed to see if it matches a device path node that the bus driver knows how to produce.

For example, a serial port can have only one device attached to it. This device may be a terminal, a mouse, or a drill press, for example. The driver that consumes the Serial I/O Protocol from a handle must create a child handle with the produced protocol that uses the services of the Serial I/O Protocol.

6.2.8.1 Sample Bus Drivers That Produce at Most One Child in Start()

Listed below are the bus drivers from the *EDK* that produce at most one child handle in a call to the **Start()** service of the **EFI_DRIVER_BINDING_PROTOCOL**.

- BiosSnp16
- Console\Terminal
- DebugPort
- IsaSerial
- Undi
- WinNtSerialIo

6.2.9 Bus Drivers That Produce No Children in Start()

If a bus controller supports hot-plug devices and the UEFI driver wants to support hot-plug events, then no child handles should be produced in **Start()**. Instead, a periodic timer event should be created, and each time the notification function for the periodic timer event is called, the bus driver should check to see if any devices have been hot added or hot removed from the bus. Any devices that were already plugged into the bus when the driver was first started will look like they were just hot added, so the child handles for the devices that were already plugged into the bus will be produced the first time the notification function is executed.

The USB bus driver is the only driver in the *EDK* that produces no children in the **Start()** service of the **EFI_DRIVER_BINDING_PROTOCOL**. This driver is in the **\EDK\Sample\Bus\Usb\UsbBus\Dxe** directory.

6.2.10 Bus Drivers That Produce Children with Multiple Parents

Sometimes a bus driver may produce a child handle, and that child handle will actually use the services of multiple parent controllers. This design is useful when a group of parent controllers needs to be multiplexed. The bus driver in this case would manage multiple parent controllers and produce a single child handle. The services produced on

that single child handle would make use of the services from each of the parent controllers. Typically, the child device is a virtual device, so a Device Path Protocol would not be installed onto the child handle.

The console splitter bus driver is the only driver in the *EDK* that produces children with multiple parent controllers in the **`Start()`** service of the **`EFI_DRIVER_BINDING_PROTOCOL`**. This driver is in the **`\Sample\Universal\Console\ConSplitter\Dxe`** directory.

6.3 Hybrid Drivers

This driver type has features of both a device driver and a bus driver. The main distinction between a device driver and a bus driver is that a bus driver creates child handles and a device driver does not create any child handles. In addition, a bus driver is allowed only to install produced protocols on the newly created child handles. A hybrid driver does the following:

- Creates new child handles.
- Installs produced protocols on the child handles.
- Installs produced protocols onto the bus controller handle.

A driver for a multi-channel RAID SCSI host controller is a hybrid driver. It produces the Ext SCSI Pass Thru Protocol (with the logical bit on) on the controller handle and creates child handles with Ext SCSI Pass Thru Protocol for each physical channel (with the logical bit off).

Section 6.3.1 below lists the hybrid drivers from the *EDK*.

6.3.1 Sample Hybrid Drivers in the EDK

- Console\Terminal
- Ide
- Usb\UsbBus
- WinNtThunk\WinNtBusDriver

6.3.2 Required Hybrid Driver Features

Hybrid drivers are required to implement the following features:

- Installs one or more instances of the **`EFI_DRIVER_BINDING_PROTOCOL`** in the driver's entry point.
- Manages one or more controller handles. Even if a driver writer is convinced the driver will manage only a single bus controller, the driver should be designed to manage multiple bus controllers. The overhead for this functionality is low, and it will make the driver more portable.
- Produces child handles. This feature is the key distinction between device drivers and bus drivers.
- Consumes one or more I/O-related protocols from a controller handle.

- Produces one or more I/O-related protocols on the same controller handle.
- Produces one or more I/O-related protocols on each child handle.

6.3.3 Optional Hybrid Driver Features

Hybrid drivers can optionally implement the following features:

Installs one or more instances of the `EFI_COMPONENT_NAME_PROTOCOL` in the driver's entry point.

Implementing this feature is strongly recommended. It allows a driver to provide human-readable names for the driver and the controllers that the driver is managing. Hybrid drivers should also provide names for the child handles created by the hybrid driver. Some platform design guides such as *DIG64* require this feature.

Installs one or more instances of the `EFI_DRIVER_CONFIGURATION_PROTOCOL` in the driver's entry point.

If a driver has any configurable options for the controller or the children, then this protocol is required. Some platform design guides such as *DIG64* require this feature.

Installs one or more instances of the `EFI_DRIVER_DIAGNOSTICS_PROTOCOL` in the driver's entry point.

If a driver wishes to provide diagnostics for the controllers for the children that the driver manages, then this protocol is required. This protocol is the only mechanism that is available to a driver when the driver wants to alert the user of a problem that was detected with a controller. Some platform design guides such as *DIG64* require this feature.

Provides an `EFI_LOADED_IMAGE_PROTOCOL.Unload()` service, so the driver can be dynamically unloaded.

It is recommended that this feature be implemented during driver development, driver debug, and system integration. It is strongly recommended that this service remain in drivers for add-in adapter to help debug interaction issues during system integration.

Installs an `EFI_DEVICE_PATH_PROTOCOL` on each child handle that is created.

This feature is required only if the child handle represents a physical device. If a child handle represents a virtual device, then an **`EFI_DEVICE_PATH_PROTOCOL`** is not required.

Creates an Exit Boot Services event in the driver's entry point.

This feature is required only if the driver is required to place the devices it manages in a specific state just before control is handed to an operating system.

Creates a Set Virtual Address Map event in the driver's entry point.

This feature is required only for a device driver that is also an EFI runtime driver.

6.4 Service Drivers

A service driver does not manage any devices and it does not produce any instances of the **EFI_DRIVER_BINDING_PROTOCOL**. It is a simple driver that produces one or more protocols on one or more new service handles. These service handles do not have a Device Path Protocol because they do not represent physical devices. The driver entry point returns **EFI_SUCCESS** after the service handles are created and the protocols are installed, which leaves the driver resident in system memory.

Section 6.4.1 below lists some service drivers from the *EDK*.

6.4.1 Sample Service Drivers in the EDK

- Bis
- DebugSupport
- Decompress
- Ebc
- FPSWA (Itanium only)

6.5 Root Bridge Drivers

A root bridge driver does not produce any instances of the **EFI_DRIVER_BINDING_PROTOCOL**. It is responsible for initializing and immediately creating physical controller handles for the root bridge controllers in a platform. The driver must install the Device Path Protocol onto the physical controller handles because the root bridge controllers represent physical devices.

Section 6.5.1 below lists the root bridge drivers from the *EDK*.

6.5.1 Sample Root Bridge Drivers in the EDK

- PciRootBridge
- WinNtThunk\WinNtPciRootBridge
- BiosInt\Disk

6.6 Initializing Drivers

An initializing driver does not create any handles and it does not add any protocols to the handle database. Instead, this type of driver performs some initialization operations and then intentionally returns an error code so the driver is unloaded from system memory. There are no drivers in the *EDK* of this type.

7 Driver Entry Point

This chapter covers the entry point for the different classes of UEFI drivers and their optional features. The most common class of UEFI driver is one that follows the UEFI Driver Model. This class of driver will be discussed first, followed by the other major types of drivers and the optional features that drivers may choose to implement. Following are the classes of UEFI drivers that will be discussed:

- Device, bus, and hybrid drivers that follows the UEFI Driver Model
- Initializing driver
- Root bridge driver
- Service driver

Note: *Unloadable drivers, Exit Boot Services events, and Set Virtual Address Map events will also be discussed.*

The driver entry point is the function that is called when an UEFI driver is started with the **`gBS->StartImage()`** service. At this point the driver has already been loaded into memory with the **`gBS->LoadImage()`**service. UEFI drivers use the PE/COFF image format that is defined in the *Microsoft Portable Executable and Common Object File Format Specification*. This format supports a single entry point in the code section of the image. The **`gBS->StartImage()`** service transfers control to the UEFI driver at this entry point.

Example 66 below shows the entry point to an UEFI driver called **`Abc`**. This example will be expanded upon as each of UEFI driver classes and features are discussed. The entry point to an UEFI driver is identical to the standard UEFI image entry point that is discussed in section 4.1 of the *UEFI 2.0 Specification*. The image handle of the UEFI driver and a pointer to the EFI System Table are passed into every UEFI driver. The image handle allows the UEFI driver to discover information about itself, and the pointer to the EFI System Table allows the UEFI driver to make EFI Boot Service and EFI Runtime Service calls. The UEFI driver can use the EFI Boot Services to access the protocol interfaces that are installed in the handle database, which allows the UEFI driver to use the services that are provided through the various protocol interfaces. The driver entry point is preceded by the macro **`EFI_DRIVER_ENTRY_POINT()`** which declares the driver entry point so source-level debugging is enabled in all build environments. Note that the Driver Entry point is also defined in the **`.inf`** build file. See section 23.1.1 for more details on **`make.inf`**.

```
EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
}
```

Example 66. Generic Entry Point

7.1 UEFI Driver Model Driver Entry Point

Drivers that follow the UEFI Driver Model are not allowed to touch any hardware in their driver entry point. In fact, these types of drivers do very little in their driver entry point. They are simply required to register interfaces in the handle database by installing protocols. This design allows these types of drivers to be loaded at any point in the system initialization sequence because their driver entry points depend only on a few of the EFI Boot Services. The protocol interfaces that are installed in the driver entry point are used at a later time to initialize, configure, or diagnose the console and boot devices that required to boot an operating system.

All UEFI drivers that follow the UEFI Driver Model must install one or more instances of the Driver Binding Protocol onto handles in the handle database. The first Driver Binding Protocol is typically installed onto the image handle for the UEFI driver. All additional instances of the Driver Binding Protocol must be installed onto other handles.

UEFI drivers may optionally support the following:

- Component Name Protocol
- Driver Configuration Protocol
- Driver Diagnostics Protocol

Some platform specifications, such as *DIG64*, require these optional protocols.

UEFI driver Library Services are provided to simplify the driver entry point of an UEFI driver. The examples in this section make use of the two UEFI Driver Library Services shown in Example 67 below. The first library function initializes the UEFI driver Library and installs the Driver Binding Protocol onto the handle specified by *DriverBindingHandle*. *DriverBindingHandle* is typically the same as *ImageHandle*, but if it is **NULL**, then the Driver Binding Protocol is installed onto a newly created handle. The second library function also initializes the UEFI driver Library and it installs all the driver-related protocols onto the handle specified by *DriverBindingHandle*. The optional driver-related protocols are defined to be **OPTIONAL** because they can be **NULL** if a driver does not wish to produce that specific optional protocol. Once again, the *DriverBindingHandle* is typically the same as *ImageHandle*, but if it is **NULL**, then all the driver-related protocols will be installed onto a newly created handle.

```
EFI_STATUS
EfiLibInstallDriverBinding (
  IN EFI_HANDLE                         ImageHandle,
  IN EFI_SYSTEM_TABLE                   *SystemTable,
  IN EFI_DRIVER_BINDING_PROTOCOL        *DriverBinding,
  IN EFI_HANDLE                         DriverBindingHandle
  );

EFI_STATUS
EfiLibInstallAllDriverProtocols (
  IN EFI_HANDLE                         ImageHandle,
  IN EFI_SYSTEM_TABLE                   *SystemTable,
  IN EFI_DRIVER_BINDING_PROTOCOL        *DriverBinding,
  IN EFI_HANDLE                         DriverBindingHandle,
  IN EFI_COMPONENT_NAME_PROTOCOL        *ComponentName,         OPTIONAL
  IN EFI_DRIVER_CONFIGURATION_PROTOCOL  *DriverConfiguration,   OPTIONAL
  IN EFI_DRIVER_DIAGNOSTICS_PROTOCOL    *DriverDiagnostics      OPTIONAL
  );
```

Example 67. Driver Library Functions

The example that follows is an example of the entry point to the Abc driver that installs the Driver Binding Protocol gAbcDriverBindingProtocol and the Component Name Protocol gAbcComponentName onto the Abc driver's image handle and does not install any of the other optional driver-related protocols. This driver simply returns the status from the UEFI driver Library function **`EfiLibInstallAllDriverProtocols()`**. See chapter 9 for details on the services and data fields that are produced by the Driver Binding Protocol.

```
EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
  AbcDriverBindingSupported,
  AbcDriverBindingStart,
  AbcDriverBindingStop,
  0x10,
  NULL,
  NULL
};

EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
  AbcComponentNameGetDriverName,
  AbcComponentNameGetControllerName,
  "eng"
};

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  //
  // Initialize a simple UEFI driver that follows the UEFI Driver Model
  //
  return EfiLibInstallAllDriverProtocols (
            ImageHandle,                // Driver's image handle
            SystemTable,                // EFI System Table Pointer
            &gAbcDriverBinding,         // Required parameters
            ImageHandle,                // Handle for driver-related
protocols
            &gAbcComponentName,         // Component Name Procol. May be
NULL.
            NULL,                       // Configuration Protocol. May be
NULL.
            NULL                        // Diagnostics Protocol. May be
NULL.
            );

}
```

Example 68. Simple UEFI Driver Model Driver Entry Point

Example 69 below shows another example of the entry point to the **Abc** driver that installs the Driver Binding Protocol **gAbcDriverBindingProtocol** onto the **Abc** driver's image handle and the optional driver-related protocols. This driver returns the status from the UEFI driver Library function **EfiLibInstallAllDriverProtocols()**. This library function is used if one or more of the optional driver related protocols are being installed.

See chapters 09, 10, 11, and 12 respectively for details on the services and data fields produced by the Driver Binding Protocol, Component Name Protocol, Driver Configuration Protocol, and Driver Diagnostics Protocol.

```
EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
  AbcDriverBindingSupported,
  AbcDriverBindingStart,
  AbcDriverBindingStop,
  0x10,
  NULL,
  NULL
};

EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
  AbcComponentNameGetDriverName,
  AbcComponentNameGetControllerName,
  "eng"
};

EFI_DRIVER_CONFIGURATION_PROTOCOL gAbcDriverConfiguration = {
  AbcDriverConfigurationSetOptions,
  AbcDriverConfigurationOptionsValid,
  AbcDriverConfigurationForceDefaults,
  "eng"
};

EFI_DRIVER_DIAGNOSTICS_PROTOCOL gAbcDriverDiagnostics = {
  AbcDriverDiagnosticsRunDiagnostics,
  "eng"
};

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  //
  // Initialize a complex UEFI driver that follows the UEFI Driver Model
  //
  return EfiLibInstallAllDriverProtocols (
            ImageHandle,                // Driver's image handle
            SystemTable,                // EFI System Table Pointer
            &gAbcDriverBinding,         // Required parameters
            ImageHandle,                // Handle for driver-related
protocols
            &gAbcComponentName,         // Component Name Procol. May be
NULL.
            &gAbcDriverConfiguration,   // Configuration Protocol. May be
NULL.
            &gAbcDriverDiagnostics      // Diagnostics Protocol. May be
NULL.
            );
}
```

Example 69. Complex UEFI Driver Model Driver Entry Point

7.1.1 Multiple Driver Binding Protocols

If an UEFI driver supports more than one parent I/O abstraction, then the driver should produce a Driver Binding Protocol for each of the parent I/O abstractions. For example, an UEFI driver could be written to support more than one type of hardware device (for example, USB and PCI). If code can be shared for the common features of a hardware device, then such a driver should save space and reduce maintenance. The only two

drivers in the *EDK* that produce more than one Driver Binding Protocol are the console platform driver and the console splitter driver. These drivers contain multiple Driver Binding Protocols because they depend on multiple console-related parent I/O abstractions.

The first Driver Binding Protocol is typically installed onto the image handle of the UEFI driver, and additional Driver Binding Protocols are installed onto new handles. The UEFI driver Library functions that were used in the previous two examples support the creation of new handles by passing in a **NULL** for the fourth argument. Example 70 below shows the driver entry point for a driver that produces two instances of the Driver Binding Protocol. The optional driver-related protocols are installed onto the image handle with the first Driver Binding Protocol. See chapters 9, 10, 11, and 12 respectively for details on the services and data fields produced by the Driver Binding Protocol, Component Name Protocol, Driver Configuration Protocol, and Driver Diagnostics Protocol.

```
EFI_DRIVER_BINDING_PROTOCOL gAbcFooDriverBinding = {
  AbcFooDriverBindingSupported,
  AbcFooDriverBindingStart,
  AbcFooDriverBindingStop,
  0x10,
  NULL,
  NULL
};

EFI_DRIVER_BINDING_PROTOCOL gAbcBarDriverBinding = {
  AbcBarDriverBindingSupported,
  AbcBarDriverBindingStart,
  AbcBarDriverBindingStop,
  0x10,
  NULL,
  NULL
};

EFI_COMPONENT_NAME_PROTOCOL gAbcFooComponentName = {
  AbcFooComponentNameGetDriverName,
  AbcFooComponentNameGetControllerName,
  "eng"
};

EFI_DRIVER_CONFIGURATION_PROTOCOL gAbcFooDriverConfiguration = {
  AbcFooDriverConfigurationSetOptions,
  AbcFooDriverConfigurationOptionsValid,
  AbcFooDriverConfigurationForceDefaults,
  "eng"
};

EFI_DRIVER_DIAGNOSTICS_PROTOCOL gAbcFooDriverDiagnostics = {
  AbcFooDriverDiagnosticsRunDiagnostics,
  "eng"
};

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS  Status;

  //
  // Install first Driver Binding Protocol and the rest of the driver-
related
  // protocols onto the driver's image handle
  //
  Status = EfiLibInstallAllDriverProtocols (
             ImageHandle,                  // Driver's image handle
             SystemTable,                  // EFI System Table Pointer
             &gAbcFooDriverBinding,        // Driver Binding Protocol
             ImageHandle,                  // Handle for driver-related
                                           // protocols
             &gAbcFooComponentName,        // Component Name Protocol
             &gAbcFooDriverConfiguration,  // Configuration Protocol
             &gAbcFooDriverDiagnostics     // Diagnostics Protocol
             );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Install second Driver Binding Protocol onto a new handle
```

```
  //
  return EfiLibInstallAllDriverProtocols (
            ImageHandle,           // Driver's image handle
            SystemTable,           // EFI System Table Pointer
            gAbcBarDriverBinding,  // Driver Binding Protocol
            NULL,                  // Handle for driver-related protocols
            NULL,                   // Component Name Protocol. May be NULL.
            NULL,                  // Configuration Protocol. May be NULL.
            NULL                   // Diagnostics Protocol. May be NULL.
            );
}
```

Example 70. Multiple Driver Binding Protocols

7.1.2 Adding the Unload Feature

Any UEFI driver can be made unloadable. This feature is useful for some driver classes, but it may not be useful at all for other driver classes. It does not make sense to add the unload feature to an initializing driver because this class of driver will return an error from the driver entry point, which will force the UEFI image Services to automatically unload the initializing driver.

It usually does not make sense for root bridge drivers or service drivers to add the unload feature. These classes of driver typically produce protocols that are consumed by other UEFI drivers to produce basic console functions and boot device abstractions. If a root bridge driver or a service driver is unloaded, then any UEFI driver that was using the protocols from those drivers would start to fail. If a root bridge driver or service driver can guarantee that it is not being used by any other EFI components, then they may be unloaded without any adverse side effects.

Peripheral device (PCI, USB, etc…) drivers should include the unload feature.

The **`Unload()`** function can be very helpful. It allows the "unload" command in the UEFI Shell to completely remove an UEFI driver image from memory and remove all of the driver's handles and protocols from the handle database. If a driver is suspected of causing a bug, it is often helpful to "unload" the driver from the UEFI Shell and then proceed to run tests knowing that the driver is no longer present in the platform. In these cases, the **`Unload()`** feature is superior to simply stopping the driver with the **`disconnect`** UEFI Shell command. If a driver is just disconnected, then the UEFI Shell commands **`connect`** and **`reconnect`** may inadvertently restart the driver.

The unload feature is also very helpful when testing and developing new versions of the driver. The old version can be completely unloaded (removed from the system) and new versions of the driver, which may have the same version number, can safely be installed in the system without concerns that the older version of the driver may get invoked during the next connect or reconnect operation.

Because **`Unload()`** completely removes the driver from system memory, it might not be possible to load it back into the system. For example, if the driver is stored in system firmware or in a PCI option ROM, no method may be available for reloading the driver without completely rebooting the system.

The **`Unload()`** service is one of the fields in the **`EFI_LOADED_IMAGE_PROTOCOL`**. This protocol is automatically created and installed when an UEFI image is loaded with the EFI Boot Service **`LoadImage()`**. When the **`EFI_LOADED_IMAGE_PROTOCOL`** is created by

LoadImage(), the **Unload()** service is initialized to **NULL**. It is the driver entry point's responsibility to register the **Unload()** function in the **EFI_LOADED_IMAGE_PROTOCOL**.

It is recommended that UEFI drivers that follow the UEFI Driver Model add the unload feature. This feature is very useful during driver development, driver debug, and system integration. It is strongly recommended that this service remain in drivers for add-in adapters to help debug interaction issues during system integration.

Example 71 below shows the same driver entry point from Example 68 with the unload feature added. Only a template for the **Unload()** function is shown in this example because the implementation of this service will vary from driver to driver. The **Unload()** service is responsible for cleaning up everything that the driver has done since it was initialized. This responsibility means that the **Unload()** service should do the following:

- Free any resources that were allocated.
- Remove any protocols that were added.
- Destroy any handles that were created.

If for some reason the **Unload()** service does not want to unload the driver at the time the **Unload()** service is called, it can return an error and the driver will not be unloaded. The only way that a driver is actually unloaded is if the **Unload()** service has been registered in the **EFI_LOADED_IMAGE_PROTOCOL** and the **Unload()** service returns **EFI_SUCCESS**.

```
EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
  AbcDriverBindingSupported,
  AbcDriverBindingStart,
  AbcDriverBindingStop,
  0x10,
  NULL,
  NULL
};

EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
  AbcComponentNameGetDriverName,
  AbcComponentNameGetControllerName,
  "eng"
};

EFI_STATUS
EFIAPI
AbcUnload (
  IN EFI_HANDLE  ImageHandle
  )
{
  return EFI_SUCCESS;
}

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS                  Status;
  EFI_LOADED_IMAGE_PROTOCOL  *LoadedImage;

  //
  // Retrieve the Loaded Image Protocol from image handle
  //
  Status = gBS->OpenProtocol (
                  ImageHandle,
                  &gEfiLoadedImageProtocolGuid,
                  (VOID **)&LoadedImage,
                  ImageHandle,
                  ImageHandle,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Fill in the Unload() service of the Loaded Image Protocol
  //
  LoadedImage->Unload = AbcUnload;

  //
  // Initialize a simple UEFI driver that follows the UEFI Driver Model
  //
  return EfiLibInstallAllDriverProtocols (
           ImageHandle,                // Driver's image handle
           SystemTable,                // EFI System Table Pointer
           &gAbcDriverBinding,         // Required parameters
           ImageHandle,                // Handle for driver related
protocols
           &gAbcComponentName,         // Component Name Procol. May be
NULL.
```

```
            NULL,                       // Configuration Protocol. May be
NULL.
            NULL                        // Diagnostics Protocol. May be
NULL.
            );
}
```

Example 71. Add the Unload Feature

Example 72 below shows one possible implementation of the **`Unload()`** function for an UEFI driver that follows the UEFI Driver Model. It finds all the devices it is managing and disconnects the driver from those devices. Then, the protocol interfaces that were installed in the driver entry point must be removed. The example shown here matches the driver entry point from Example 69. There are many possible algorithms that can be implemented in the unload service. A driver may choose to be unloadable if and only if it is not managing any devices at all. A driver may also choose to keep track of the devices it is managing internally, so it can selectively disconnect itself from those devices when it is unloaded.

```
EFI_STATUS
EFIAPI
AbcUnload (
  IN EFI_HANDLE  ImageHandle
  )
{
  EFI_STATUS  Status;
  EFI_HANDLE  *DeviceHandleBuffer;
  UINTN       DeviceHandleCount;
  UINTN       Index;

  //
  // Get the list of all the handles in the handle database.
  // If there is an error getting the list, then the unload operation
fails.
  //
  Status = gBS->LocateHandleBuffer (
                AllHandles,
                NULL,
                NULL,
                &DeviceHandleCount,
                &DeviceHandleBuffer
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Disconnect the driver specified by ImageHandle from all the devices
in the
  // handle database.
  //
  for (Index = 0; Index < DeviceHandleCount; Index++) {
    Status = gBS->DisconnectController (
                  DeviceHandleBuffer[Index],
                  ImageHandle,
                  NULL
                  );
  }

  //
  // Free the buffer containing the list of handles from the handle
database
  //
  if (DeviceHandleBuffer != NULL) {
    gBS->FreePool(DeviceHandleBuffer);
  }

  //
  // Uninstall all the protocols that were installed in the driver entry
point
  //
  Status = gBS->UninstallMultipleProtocolInterfaces (
                  ImageHandle,
                  &gEfiDriverBindingProtocolGuid, &gAbcDriverBinding,
                  &gEfiComponentNameProtocolGuid, &gAbcComponentName,
                  &gEfiDriverConfigurationProtocolGuid,
&gAbcDriverConfiguration,
                  &gEfiDriverDiagnosticsProtocolGuid,
&gAbcDriverDiagnostics
                  NULL
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Do any additional cleanup that is required for this driver
```

```
  //
  return EFI_SUCCESS;
}
```

Example 72. Unload Function

7.1.3 Adding the Exit Boot Services Feature

Some UEFI drivers may need to put their devices into a known state prior to booting an operating system. This case is considered to be very rare because the OS-present drivers should not depend on an UEFI driver running at all. Not depending on a running UEFI driver means that an OS-present driver should be able to handle the following:

- A device in its power-on reset state
- A device that was recently hot added while the OS is running
- A device that was managed by an UEFI driver up to the point where the OS was booted
- A device that was managed for a short period of time by an UEFI driver

None of the drivers in the *EDK* use the feature described here. It is documented here to show what is possible if this feature is ever required.

In the rare case where an UEFI driver is required to place a device in a known state before booting an operating system, the driver can use a special event type called an *Exit Boot Services event.* This event is signaled when an OS loader or OS kernel calls the EFI Boot Service **`gBS->ExitBootServices()`**. This call is the point in time where the system firmware still owns the platform, but the system firmware is just about to transfer system ownership to the operating system. In this transition time, no modifications to the EFI memory map are allowed (see section 5.1 of the *UEFI 2.0 Specification*). This requirement means that the notification function for an Exit Boot Services event is not allowed to directly or indirectly allocate or free and memory through the EFI Memory Services.

Example 73 below shows the same example from Example 68, but an Exit Boot Services event is also created. The template for the notification function for the Exit Boot Services event is also shown. This notification function will typically contain code to find the list of device handles that the driver is currently managing, and it will then perform operations on those handles to make sure they are in the proper OS handoff state. Remember that no memory allocation or free operations can be performed from this notification function.

```
EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
  AbcDriverBindingSupported,
  AbcDriverBindingStart,
  AbcDriverBindingStop,
  0x10,
  NULL,
  NULL
};

EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
  AbcComponentNameGetDriverName,
  AbcComponentNameGetControllerName,
  "eng"
};

VOID
EFIAPI
AbcNotifyExitBootServices (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  //
  // Put driver-specific actions here.
  // No EFI Memory Service may be used directly or indirectly.
  //
}

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS  Status;
  EFI_EVENT   *ExitBootServicesEvent;

  //
  // Create an Exit Boot Services event.
  //
  Status = gBS->CreateEvent (
                  EFI_EVENT_SIGNAL_EXIT_BOOT_SERVICES,
                  EFI_TPL_NOTIFY,
                  AbcNotifyExitBootServices,
                  NULL,
                  &ExitBootServicesEvent
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Initialize a simple UEFI driver that follows the UEFI Driver Model
  //
  return EfiLibInstallAllDriverProtocols (
           ImageHandle,                // Driver's image handle
           SystemTable,                // EFI System Table Pointer
           &gAbcDriverBinding,         // Required parameters
           ImageHandle,                // Handle for driver-related
protocols
           &gAbcComponentName,         // Component Name Procol. May be
NULL.
           NULL,                       // Configuration Protocol. May be
NULL.
```

```
            NULL                        // Diagnostics Protocol. May be
NULL.
            );

}
```

Example 73. Adding the Exit Boot Services Feature

If an UEFI driver supports both the unload feature and the Exit Boot Services feature, then the **`Unload()`** function must destroy the Exit Boot Services event by calling **`gBS->CloseEvent()`**. In this case, the Exit Boot Services event would likely be declared as a global variable, so the event could easily be destroyed in the **`Unload()`** function. Example 74 below shows a driver that supports both the unload service and an Exit Boot Services event.

```
EFI_EVENT  *gExitBootServicesEvent;

VOID
EFIAPI
AbcNotifyExitBootServices (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  //
  // Put driver-specific actions here.
  // No EFI Memory Service may be used directly or indirectly.
  //
}

EFI_STATUS
EFIAPI
AbcUnload (
  IN EFI_HANDLE  ImageHandle
  )
{
  gBS->CloseEvent (gExitBootServicesEvent);
}

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS                  Status;
  EFI_LOADED_IMAGE_PROTOCOL  *LoadedImage;

  //
  // Initialize the UEFI driver Library
  //
  EfiInitializeDriverLib (ImageHandle, SystemTable);

  //
  // Create an Exit Boot Services event.
  //
  Status = gBS->CreateEvent (
                EFI_EVENT_SIGNAL_EXIT_BOOT_SERVICES,
                EFI_TPL_NOTIFY,
                AbcNotifyExitBootServices,
                NULL,
                &gExitBootServicesEvent
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Retrieve the Loaded Image Protocol from image handle
  //
  Status = gBS->OpenProtocol (
                ImageHandle,
                &gEfiLoadedImageProtocolGuid,
                (VOID **)&LoadedImage,
                ImageHandle,
                ImageHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
  if (EFI_ERROR (Status)) {
    return Status;
```

```
  }

  //
  // Fill in the Unload() service of the Loaded Image Protocol
  //
  LoadedImage->Unload = AbcUnload;

  return EFI_SUCCESS;
}
```

Example 74. Add the Unload and Exit Boot Services Event Feature

7.2 Initializing Driver Entry Point

Example 75 below shows a template for an initializing driver called **Abc**. This driver initializes one or more components in the platform and exits. It does not produce any services that are required after the entry point has been executed. This type of driver will return an error from the entry point, so the driver will be unloaded by the UEFI image Services. An initializing driver will never produce an **Unload()** service because they are always unloaded after their entry point is executed. This type of driver is not used by IHVs. Instead, it is typically used by OEMs and IBVs to initialize the state of a hardware component in the platform such as the processor or chipset components.

```
EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  //
  // Initialize the UEFI driver Library
  //
  EfiInitializeDriverLib (ImageHandle, SystemTable);

  //
  // Perform some platform initialization operations here
  //

  return EFI_LOAD_ERROR;
}
```

Example 75. Initializing Driver Entry Point

7.3 Service Driver Entry Point

A service driver produces one or more protocol interfaces on the driver's image handle or on newly created handles. Example 76 below shows the Decompress Protocol being installed onto the driver's image handle. A service driver may produce an **Unload()** service, and that service would be required to uninstall the protocols that were installed in the driver's entry point. The **Unload()** service can be a dangerous operation because there is no way for the service driver to know if the protocols that it installed are being used by other EFI components. If the service driver is unloaded and

other EFI components are still using the protocols that were produced by the unloaded driver, then the system will likely fail.

```
EFI_DECOMPRESS_PROTOCOL gAbcDecompress = {
  GetInfo,
  Decompress
};

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  //
  // Initialize the UEFI driver Library
  //
  EfiInitializeDriverLib (ImageHandle, SystemTable);

  //
  // Install the Decompress Protocol onto the driver's image handle
  //
  return gBS->InstallMultipleProtocolInterfaces (
                &ImageHandle,
                &gEfiDecompressProtocolGuid, &gAbcDecompress,
                NULL
                );
}
```

Example 76. Service Driver Entry Point – Image Handle

A service driver may also install its protocol interfaces onto one or more new handles in the handle database. Example 77 below shows a template for a service driver called **Abc** that produces the Decompress Protocol on a new handle.

```
EFI_DECOMPRESS_PROTOCOL gAbcDecompress = {
  GetInfo,
  Decompress
};

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_HANDLE  NewHandle;

  //
  // Initialize the UEFI driver Library
  //
  EfiInitializeDriverLib (ImageHandle, SystemTable);

  //
  // Install the Decompress Protocol onto the a new handle
  //
  NewHandle = NULL;
  return gBS->InstallMultipleProtocolInterfaces (
                &NewHandle,
                &gEfiDecompressProtocolGuid, &gAbcDecompress,
                NULL
                );
}
```

Example 77. Service Driver Entry Point – New Handle

7.4 Root Bridge Driver Entry Point

Root bridge drivers produce handles and software abstractions for the bus types that are directly produced by a core chipset. The PCI Root Bridge I/O Protocol is the software abstraction for root bridges that is defined in the *UEFI 2.0 Specification.* UEFI drivers that produce this protocol do not follow the UEFI Driver Model. Instead, they initialize hardware and directly produce the handles and protocols in the driver entry point. Root bridge drivers are slightly different from *service drivers* in the following ways:

- Root bridge drivers always create new handles.
- The root bridge driver is responsible for installing both of the following:
  - The software abstraction, such as the PCI Root Bridge I/O Protocol
  - The Device Path Protocol that describes the programmatic path to the root bridge device

Example 78 below shows a template for a root bridge driver that produces one handle for a system with a single PCI root bridge. A Device Path Protocol with an ACPI device path node and the PCI Root Bridge I/O Protocol are installed onto a newly created handle. The ACPI device path node for the PCI root bridge must match the description of the PCI root bridge in the ACPI table for the platform. In this example, the Device Path Protocol and PCI Root Bridge I/O Protocol are declared as global variables. An actual driver may need to keep track of additional private data fields to manage a PCI root bridge properly. In that case, global variables would not be used. Instead, a

constructor function would allocate and initialize the public and private data fields. This concept is described in more detail in chapter 8.

```
typedef struct {
  ACPI_HID_DEVICE_PATH      AcpiDevicePath;
  EFI_DEVICE_PATH_PROTOCOL  EndDevicePath;
} EFI_PCI_ROOT_BRIDGE_DEVICE_PATH;

EFI_PCI_ROOT_BRIDGE_DEVICE_PATH  gAbcPciRootBridgeDevicePath = {
  {
    ACPI_DEVICE_PATH,                                   // Type
    ACPI_DP,                                            // Subtype
    (UINT8) (sizeof(ACPI_HID_DEVICE_PATH)),             // Length (lower 8
bits)
    (UINT8) ((sizeof(ACPI_HID_DEVICE_PATH)) >> 8),      // Length (upper 8
bits)
    EISA_PNP_ID(0x0A03),                                // HID
    0                                                   // UID
  },
  {
    END_DEVICE_PATH_TYPE,                               // Type
    END_ENTIRE_DEVICE_PATH_SUBTYPE,                     // Subtype
    END_DEVICE_PATH_LENGTH,                             // Length (lower 8
bits)
    0                                                   // Length (upper 8
bits)
  }
};

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL gAbcPciRootBridgeIo = {
  NULL,                                 // ParentHandle
  AbcPciRootBridgeIoPollMem,            // PollMem()
  AbcPciRootBridgeIoPollIo,             // PolIo()
  {
    AbcPciRootBridgeIoMemRead,          // Mem.Read()
    AbcPciRootBridgeIoMemWrite          // Mem.Write()
  },
  {
    AbcPciRootBridgeIoIoRead,           // Io.Read()
    AbcPciRootBridgeIoIoWrite,          // Io.Write()
  },
  {
    AbcPciRootBridgeIoPciRead,          // Pci.Read()
    AbcPciRootBridgeIoPciWrite,         // Pci.Write()
  },
  AbcPciRootBridgeIoCopyMem,            // CopyMem()
  AbcPciRootBridgeIoMap,                // Map()
  AbcPciRootBridgeIoUnmap,              // Unmap()
  AbcPciRootBridgeIoAllocateBuffer,     // AllocateBuffer()
  AbcPciRootBridgeIoFreeBuffer,         // FreeBuffer()
  AbcPciRootBridgeIoFlush,              // Flush()
  AbcPciRootBridgeIoGetAttributes,      // GetAttributes()
  AbcPciRootBridgeIoSetAttributes,      // SetAttributes()
  AbcPciRootBridgeIoConfiguration,      // Configuration()
  0                                     // SegmentNumber
};

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS  Status;
  EFI_HANDLE  NewHandle;

  //
  // Initialize the UEFI driver Library
```

```
  //
  EfiInitializeDriverLib (ImageHandle, SystemTable);

  //
  // Perform root bridge initialization operations here
  //

  //
  // Install the Device Path Protocol and PCI Root Bridge I/O Protocol
onto
  // a new handle.
  //
  NewHandle = NULL;
  Status = gBS->InstallMultipleProtocolInterfaces (
                  &NewHandle,
                  &gEfiDevicePathProtocolGuid,
&gAbcPciRootBridgeDevicePath,
                  &gEfiPciRootBridgeIoProtocolGuid, &gAbcPciRootBridgeIo,
                  NULL
                  );
  return Status;
}
```

Example 78. Single PCI Root Bridge Driver Entry Point

Example 79 below shows a template for a root bridge driver that produces four handles for a system with four PCI root bridges. A Device Path Protocol with an ACPI device path node and the PCI Root Bridge I/O Protocol are installed onto each of the newly created handles. The ACPI device path nodes for each of the PCI root bridges must match the description of the PCI root bridges in the ACPI tables for the platform. In this example, the _UID field for the root bridges has the values of 0, 1, 2, and 3. However, there is no requirement that the _UID field starts at 0 or that they are contiguous. The only requirement is that the _UID field for each root bridge matches the _UID field in the ACPI table that describes the same root bridge controller. The Device Path Protocol and PCI Root Bridge I/O Protocol are declared as global variables, and copies of those global variables are made for each PCI root bridge. An actual driver may need to keep track of additional private data fields to manage each PCI root bridge properly. In that case, global variables would not be used. Instead, a constructor function would allocate and initialize the public and private data fields. This concept is described in more detail in chapter 8.

```
#define HWP_EISA_ID_CONST  0x22F0
#define COMPRESSED_ASCII (C1, C2, C3) ((((C1) - 'A') & 0x1f) << 10) | \
                                      ((((C2) - 'A') & 0x1f) << 5 ) | \
                                      ((((C3) - 'A') & 0x1f)

typedef struct {
  ACPI_ EXTENDED_HID_DEVICE_PATH  AcpiDevicePath;
  EFI_DEVICE_PATH_PROTOCOL        EndDevicePath;
} EFI_PCI_ROOT_BRIDGE_DEVICE_PATH;

EFI_PCI_ROOT_BRIDGE_DEVICE_PATH  gAbcPciRootBridgeDevicePath = {
  {
    ACPI_DEVICE_PATH,                                   // Type
    ACPI_DP,                                            // Subtype
    (UINT8) (sizeof(ACPI_HID_DEVICE_PATH)),             // Length (lower 8
bits)
    (UINT8) ((sizeof(ACPI_HID_DEVICE_PATH)) >> 8),      // Length (upper 8
bits)
    EISA_ID (COMPRESSED_ASCII ('H','W','P'), 2),        // HID
    0,                                                  // UID
    EISA_PNP_ID (0x0A03)                                // CID
  },
  {
    END_DEVICE_PATH_TYPE,                               // Type
    END_ENTIRE_DEVICE_PATH_SUBTYPE,                     // Subtype
    END_DEVICE_PATH_LENGTH,                             // Length (lower 8
bits)
    0                                                   // Length (upper 8
bits)
  }
};

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL gAbcPciRootBridgeIo = {
  NULL,                                 // ParentHandle
  AbcPciRootBridgeIoPollMem,            // PollMem()
  AbcPciRootBridgeIoPollIo,             // PolIo()
  {
    AbcPciRootBridgeIoMemRead,          // Mem.Read()
    AbcPciRootBridgeIoMemWrite          // Mem.Write()
  },
  {
    AbcPciRootBridgeIoIoRead,           // Io.Read()
    AbcPciRootBridgeIoIoWrite,          // Io.Write()
  },
  {
    AbcPciRootBridgeIoPciRead,          // Pci.Read()
    AbcPciRootBridgeIoPciWrite,         // Pci.Write()
  },
  AbcPciRootBridgeIoCopyMem,            // CopyMem()
  AbcPciRootBridgeIoMap,                // Map()
  AbcPciRootBridgeIoUnmap,              // Unmap()
  AbcPciRootBridgeIoAllocateBuffer,     // AllocateBuffer()
  AbcPciRootBridgeIoFreeBuffer,         // FreeBuffer()
  AbcPciRootBridgeIoFlush,              // Flush()
  AbcPciRootBridgeIoGetAttributes,      // GetAttributes()
  AbcPciRootBridgeIoSetAttributes,      // SetAttributes()
  AbcPciRootBridgeIoConfiguration,      // Configuration()
  0                                     // SegmentNumber
};

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
```

```
{
  EFI_STATUS                        Status;
  UINTN                             Index;
  EFI_HANDLE                        NewHandle;
  EFI_PCI_ROOT_BRIDGE_DEVICE_PATH   *DevicePath;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *PciRootBridgeIo;

  //
  // Initialize the UEFI driver Library
  //
  EfiInitializeDriverLib (ImageHandle, SystemTable);

  //
  // Perform root bridge initialization operations here
  //

  //
  // Install the Device Path Protocol and PCI Root Bridge I/O Protocol
onto
  // a new handle.
  //
  for (Index = 0; Index < 4 ; Index++) {

    //
    // Make a copy of the device path and update the UID field of the
ACPI
    // device path node
    //
    DevicePath = EfiLibAllocatePool (sizeof
(EFI_PCI_ROOT_BRIDGE_DEVICE_PATH));
    if (DevicePath == NULL) {
      return EFI_OUT_OF_RESOURCES;
    }
    gBS->CopyMem (
           DevicePath,
           &gAbcPciRootBridgeDevicePath,
           sizeof (EFI_PCI_ROOT_BRIDGE_DEVICE_PATH)
           );
    DevicePath->AcpiDevicePath.UID = Index;

    //
    // Make a copy of the PCI Root Bridge I/O Protocol
    //
    PciRootBridgeIo = EfiLibAllocatePool (
                        sizeof (EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL)
                        );
    if (PciRootBridgeIo == NULL) {
      return EFI_OUT_OF_RESOURCES;
    }
    gBS->CopyMem (
           PciRootBridgeIo,
           &gAbcPciRootBridgeIo,
           sizeof (EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL)
           );

    //
    // Install the Device Path Protocol and the PCI Root Bridge I/O
Protocol
    // onto a new handle.
    //
    NewHandle = NULL;
    Status = gBS->InstallMultipleProtocolInterfaces (
                    &NewHandle,
                    &gEfiDevicePathProtocolGuid,  DevicePath,
                    &gEfiPciRootBridgeIoProtocolGuid, PciRootBridgeIo,
                    NULL
                    );
    if (EFI_ERROR (Status)) {
```

```
      return Status;
    }
  }
  return EFI_SUCCESS;
}
```

Example 79. Multiple PCI Root Bridge Driver Entry Point

7.5 Runtime Drivers

Any UEFI driver can be modified to be a runtime driver. This ability applies to the following:

- UEFI drivers that follow the UEFI Driver Model
- Initializing drivers
- Service drivers
- Root bridge drivers

However, it does not make sense to add the runtime feature to an initializing driver, because the initializing driver is unloaded after its entry point has been executed. The best example of a runtime driver that follows the UEFI Driver Model is an UNDI driver that provides services for a network interface controller (NIC).

A runtime driver is a driver that is required to provide services after **gBS->ExitBootServices()** has been called. Drivers of this type are much more difficult to implement and validate because they are required to execute in both the preboot environment where the system firmware owns the platform and while an OS is running where the OS owns the platform. The OS may choose to switch all runtime services from physical mode addressing to virtual mode addressing. The driver cannot know which type of OS is going to be booted, so the runtime driver must always be able to switch from physical addressing to virtual addressing if **gRT->SetVirtualAddressMap()** is called by an OS loader or an OS kernel. In addition, because all memory regions that are marked as boot services memory in the EFI memory map are converted to available memory when an OS is booted, a runtime driver must allocate all of its memory buffers from runtime memory.

A runtime driver will typically create the following two special events to help with these issues:

- Exit Boot Services event
- Set Virtual Address Map event

The Exit Boot Services event is signaled when the OS loader or OS kernel calls **gBS->ExitBootServices()**. After this point, the UEFI driver is not allowed to use any of the EFI Boot Services. The EFI Runtime Services and services from other runtime drivers are still available. The Set Virtual Address Map event is signaled when the OS loader or OS kernel calls **gRT->SetVirtualAddresMap()**. If this event is signaled, then the OS loader or OS kernel is requesting that all runtime components be converted from their physical address mapping to the virtual address mappings that are passed to **gRT->SetVirtualAddressMap()**. The EFI firmware does most of the work here by relocating all the UEFI images from their physically addressed code and data segments to their virtually addressed code and data segments. However, the EFI firmware cannot know what memory buffers a runtime component has allocated and what pointer

values stored within those runtime memory buffers need to be converted from their physical addresses to their virtual addresses. The notification function for the Set Virtual Address Map event is required to use the **`gRT->ConvertPointer()`** service to convert all pointers in private data structures from their physical address to their virtual addresses. This code is complex and difficult to get correct because no tools are available at this time to help know when all the pointers have been converted. The only symptom that is seen when it is not done correctly is that the OS will crash in the middle of a call to a service produced by a runtime driver.

Example 80 below shows the driver entry point for a runtime driver that creates an Exit Boot Services event *and* a Set Virtual Address Map event. The notification function for the Exit Boot Services event sets a global variable to **`TRUE`**, so the code in other functions can know if the EFI Boot Services are available. This global variable is initialized to **`FALSE`** in its declaration. The notification function for the Set Virtual Address Map event converts one global pointer from a physical address to a virtual address as an example. A real driver might have many more pointers to convert. In general, a runtime driver should be designed to reduce or eliminate pointers that need to be converted to minimize the likelihood of missing a pointer conversion.

```
VOID     *gGlobalPointer;
BOOLEAN  gAtRuntime = FALSE;

VOID
EFIAPI
AbcNotifySetVirtualAddressMap (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  gRT->ConvertPointer (
         EFI_OPTIONAL_POINTER,
         (VOID **)&gGlobalPointer
         );
}

VOID
EFIAPI
AbcNotifyExitBootServices (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  gAtRuntime = TRUE;
}

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS  Status;
  EFI_EVENT   *ExitBootServicesEvent;
  EFI_EVENT   *SetVirtualAddressMapEvent;

  //
  // Create an Exit Boot Services event.
  //
  Status = gBS->CreateEvent (
                  EFI_EVENT_SIGNAL_EXIT_BOOT_SERVICES,
                  EFI_TPL_NOTIFY,
                  AbcNotifyExitBootServices,
                  NULL,
                  &ExitBootServicesEvent
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Create a Set Virtual Address Map event.
  //
  Status = gBS->CreateEvent (
                  EFI_EVENT_SIGNAL_SET_VIRTUAL_ADDRESS_MAP,
                  EFI_TPL_NOTIFY,
                  AbcNotifySetVirtualAddressMap,
                  NULL,
                  &SetVirtualAddressMapEvent
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
```

```
  // Perform additional driver initialization here
  //

  return EFI_SUCCESS;
}
```

Example 80. Runtime Driver Entry Point

If a runtime driver also supports the unload feature, then the **`Unload()`** function must destroy the Exit Boot Services event and the Set Virtual Address Map event by calling **`gBS->CloseEvent()`**. In this case, these events would likely be declared as global variables, so the events could easily be destroyed in the **`Unload()`** function. Example 81 below shows an unloadable runtime driver.

```
EFI_EVENT  *gExitBootServicesEvent;
EFI_EVENT  *gSetVirtualAddressMapEvent;
VOID       *gGlobalPointer;
BOOLEAN    gAtRuntime = FALSE;

VOID
EFIAPI
AbcNotifySetVirtualAddressMap (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  gRT->ConvertPointer (
         EFI_OPTIONAL_POINTER,
         (VOID **)&gGlobalPointer
         );
}

VOID
EFIAPI
AbcNotifyExitBootServices (
  IN EFI_EVENT  Event,
  IN VOID       *Context
  )
{
  gAtRuntime = TRUE;
}

EFI_STATUS
EFIAPI
AbcUnload (
  IN EFI_HANDLE  ImageHandle
  )
{
  gBS->CloseEvent (gExitBootServicesEvent);
  gBS->CloseEvent (gSetVirtualAddressMapEvent);
}

EFI_DRIVER_ENTRY_POINT (AbcDriverEntryPoint)

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS                  Status;
  EFI_LOADED_IMAGE_PROTOCOL  *LoadedImage;

  //
  // Create an Exit Boot Services event.
  //
  Status = gBS->CreateEvent (
                  EFI_EVENT_SIGNAL_EXIT_BOOT_SERVICES,
                  EFI_TPL_NOTIFY,
                  AbcNotifyExitBootServices,
                  NULL,
                  &gExitBootServicesEvent
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Create a Set Virtual Address Map event.
  //
  Status = gBS->CreateEvent (
```

```
                  EFI_EVENT_SIGNAL_SET_VIRTUAL_ADDRESS_MAP,
                  EFI_TPL_NOTIFY,
                  AbcNotifySetVirtualAddressMap,
                  NULL,
                  &gSetVirtualAddressMapEvent
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Retrieve the Loaded Image Protocol from image handle
  //
  Status = gBS->OpenProtocol (
                  ImageHandle,
                  &gEfiLoadedImageProtocolGuid,
                  (VOID **)&LoadedImage,
                  ImageHandle,
                  ImageHandle,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Fill in the Unload() service of the Loaded Image Protocol
  //
  LoadedImage->Unload = AbcUnload;

  //
  // Perform additional driver initialization here
  //

  return EFI_SUCCESS;
}
```

Example 81. Runtime Driver Entry Point with Unload Feature

8 Private Context Data Structures

UEFI drivers that manage more than one controller need to be designed with reentrancy in mind. This means that the global variables should not be used to track information about individual controllers. Instead, data structures should be allocated with the EFI Memory Services for each controller, and those data structures should contain all the information that the driver requires to manage each individual controller. This chapter will introduce some object-oriented programming techniques that can be applied to drivers that manage controllers. These techniques can simplify the driver design and implementation. The concept of a private context data structure that contains all the information that is required to manage a controller will be introduced. This data structure contains the public data fields, public services, private data fields, and private services that an UEFI driver may require to manage a controller.

Some classes of UEFI drivers do not require the use of these data structures. If an UEFI driver only produces a single protocol or it manages at most one device, then the techniques presented here are not required. An initializing driver does not produce any services and does not manage any devices, so it will not use this technique. A service driver that produces a single protocol and does not manage any devices likely will not use this technique. A root bridge driver that manages a single root bridge device likely will not use this technique, but a root bridge driver that manages more than one root bridge device should use this technique. Finally, all UEFI drivers that follow the UEFI Driver Model should use this technique. Even if the driver writer is convinced that the UEFI driver will manage only a single device in a platform, this technique should still be used because it will simplify the process of updating the driver to manage more than one device. The driver writer should make as few device and platform assumptions as possible when designing a new driver.

It is possible to use other techniques to track the information that is required to manage multiple controllers in a re-entrant-safe manner, but those techniques will likely require more overhead in the driver itself to manage this information. The techniques presented here are intended to produce small driver executables, and these techniques are used throughout the drivers in the *EDK*.

8.1 Containing Record Macro

The containing record macro, which is called **`CR()`**, enables good object-oriented programming practices. It returns a pointer to the structure using a pointer to one of the structure's fields. UEFI drivers that produce protocols use this macro to retrieve the private context data structure from a pointer to a produced protocol interface. Protocol functions are required to pass in a pointer to the protocol instance as the first argument to the function. C++ does this automatically, and the pointer to the object instance is called a *`this`* pointer. Since UEFI drivers are written in C, a close equivalent is implemented by requiring that the first argument of every protocol function be the

pointer to the protocol's instance structure called *This*. Each protocol function then uses the **CR()** macro to retrieve a pointer to the private context data structure from this first argument called *This*.

Example 82 is the definition of the **CR()** macro. The **CR()** macro is provided a pointer to the following:

- A field in a data structure
- The name of the field

It uses this information to compute the offset of the field in the data structure and subtracts this offset from the pointer to the field. This calculation results in a pointer to the data structure that contains the specified field. **_CR()** returns a pointer to the data structure that contains the specified field. For debug builds, **CR()** also does an additional check to make sure the signature matches. If the signature does not match, then an **ASSERT()** message is generated and the system is haled. For production builds, the signature checks are skipped. Most UEFI drivers define additional macros based on the **CR()** macro that retrieve the private context data structure based on a *This* pointer to a produced protocol.

```
#define _CR(Record, TYPE, Field)  \
  ((TYPE *) ( (CHAR8 *)(Record) - (CHAR8 *) &(((TYPE *) 0)->Field)))

#ifdef EFI_DEBUG

  #define CR(record, TYPE, field, signature)                          \
            _CR(record, TYPE, field)->Signature != signature ?         \
            (TYPE *) (_DEBUG_ASSERT("CR has Bad Signature"), record) : \
            _CR(record, TYPE, field)

#else

  #define CR(Record, TYPE, Field, Signature)   \
            _CR(Record, TYPE, Field)

#endif
```

Example 82. Containing Record Macro Definitions

8.2 Data Structure Design

Proper data structure design is one of the keys to making drivers both simple and easy to maintain. If a driver writer fails to include fields in a private context data structure, then it may require a complex algorithm to retrieve the required data through the various EFI services. By designing in the proper fields, these complex algorithms can be avoided and the driver will have a smaller executable footprint. Static data and commonly accessed data and services that are related to the management of a device should be placed in a private context data structure. Another requirement is that the private context data structure must be easy to find when an I/O service that is produced by the driver is called. The I/O services that are produced by a driver are exported through protocol interfaces, and all protocol interface include a *This* parameter as the first argument. The *This* parameter is a pointer to the protocol interface that contains the I/O service being called. The data structure design presented here will show how the *This* pointer that is passed into an I/O service can be used to very easily gain access to the private context data structure.

The following driver types will typically use private context data structures:

- Root bridge drivers
- Device drivers
- Bus drivers
- Hybrid drivers

Hybrid drivers may even use two different private context data structures—one for the bus controller and one for the child controllers it produces. A private context data structure is typically composed of the following types of fields:

- A signature for the data structure
- The handle of the controller or the child that is being managed or produced
- The group of protocol interfaces that are being consumed
- The group of protocol interfaces that are being produced
- Private data fields and services that are used to manage a specific controller

The signature is useful when debugging UEFI drivers. These signatures are composed of four ASCII characters. When memory dumps are performed, these signatures stand out, so the beginning of specific data structures can be identified. Memory dump tools with search capabilities can also be used to find specific private context data structures in memory. In addition, debug builds of UEFI drivers can perform signature checks whenever these private context data structures are accessed. If the signature does not match, then an **`ASSERT()`** can be generated. If one of these **`ASSERT()`** messages are observed, then an UEFI driver was likely passed in a bad or corrupt memory pointer.

Device drivers will typically store the handle of the device they are managing in a private context data structure. This mechanism provides quick access to the device handle if it is needed during I/O operations or driver-related operations. Root bridge drivers and bus drivers will typically store the handle of the child that was created, and a hybrid driver will typically store both the handle of the bus controller and the handle of the child controller that was produced.

The group of consumed protocol interfaces is simply a set of pointers to the protocol interfaces that are opened in the **`Start()`** function of the driver's **`EFI_DRIVER_BINDING_PROTOCOL`**. As each protocol interface is opened using **`gBS->OpenProtocol()`**, a pointer to the consumed protocol interface is stored in the private context data structure. These same protocols must be closed in the **`Stop()`** function of the driver's **`EFI_DRIVER_BINDING_PROTOCOL`** with calls to **`gBS->CloseProtocol()`**.

The group of produced protocol interfaces declares the storage for the protocols that the driver produces. These protocols typically provide software abstractions for console or boot devices.

The number and types of the private data fields vary from driver to driver. These fields contain the context information for a device that is not contained in the consumed or produced protocols. For example, a driver for a disk device may store information about the geometry of the disk such as the number of cylinders, number of heads, and number of sectors on the physical disk that the driver is managing.

Example 83 below shows a generic template for a private context data structure that can be used for root bridge drivers, device drivers, bus drivers, or hybrid drivers. The **#define** statement at the beginning is used to initialize the *Signature* field when the private context data structure is allocated. This same **#define** statement is used to verify the *Signature* field whenever a driver accesses the private context data structure. The data structure itself contains the following:

- Signature
- Handle of the device being managed
- Pointers to the consumed protocols
- Storage for the produced protocols
- Any additional private data fields that are required to manage the device

The last part of this figure contains a set of macros that help retrieve a pointer to the private context data structure from a *This* pointer for each of the produced protocols. These macros are the simple mechanism that allows the private data fields to be accessed from the services in each of the produced protocols.

```
#define <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE
EFI_SIGNATURE_32('A','B','C','D')

typedef struct {
  UINTN                   Signature;
  EFI_HANDLE              Handle;

  //
  // Pointers to consumed protocols
  //
  EFI_<<PROTOCOL_NAME_C1>>_PROTOCOL  *<<ProtocolNameC1>>;
  EFI_<<PROTOCOL_NAME_C2>>_PROTOCOL  *<<ProtocolNameC2>>;
  // . . .
  EFI_<<PROTOCOL_NAME_Cn>>_PROTOCOL  *<<ProtocolNameCn>>;

  //
  // Produced protocols
  //
  EFI_<<PROTOCOL_NAME_P1>>_PROTOCOL  <<ProtocolNameP1>>;
  EFI_<<PROTOCOL_NAME_P2>>_PROTOCOL  <<ProtocolNameP2>>;
  // . . .
  EFI_<<PROTOCOL_NAME_Pm>>_PROTOCOL  <<ProtocolNamePm>>;

  //
  // Private functions and data fields
  //
} <<DRIVER_NAME>>_PRIVATE_DATA;

#define <<DRIVER_NAME>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_P1>>_THIS(a)  \
  CR(                                                                  \
    a,                                                                 \
    <<DRIVER_NAME>>_PRIVATE_DATA,                                      \
    <<ProtocolNameP1>>,                                                \
    <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE                             \
  )

#define <<DRIVER_NAME>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_P2>>_THIS(a)  \
  CR(                                                                  \
    a,                                                                 \
    <<DRIVER_NAME>>_PRIVATE_DATA,                                      \
    <<ProtocolNameP2>>,                                                \
    <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE                             \
  )

// . . .

#define <<DRIVER_NAME>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Pm>>_THIS(a)  \
  CR(                                                                  \
    a,                                                                 \
    <<DRIVER_NAME>>_PRIVATE_DATA,                                      \
    <<ProtocolNamePm>>,                                                \
    <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE                             \
  )
```

Example 83. Private Context Data Structure Template

Example 84 below shows an example of the private context data structure from the disk I/O driver that is listed in Appendix D. It contains the **#define** statement for the data structure's signature. In this case, the signature is the ASCII string "dskI." It also contains a pointer to the only protocol that this driver consumes, which is the Block I/O Protocol, and it contains storage for the only protocol that this driver produces, which is the Disk I/O Protocol. It also has one private data field that caches the size of the blocks that the Block I/O Protocol supports. The macro at the bottom retrieves the private context data structure from a pointer to the *DiskIo* field.

```
#define DISK_IO_PRIVATE_DATA_SIGNATURE  EFI_SIGNATURE_32 ('d','s','k','I')

typedef struct {
  UINTN                 Signature;
  EFI_DISK_IO_PROTOCOL  DiskIo;
  EFI_BLOCK_IO_PROTOCOL *BlockIo;
  UINT32                BlockSize;
} DISK_IO_PRIVATE_DATA;

#define DISK_IO_PRIVATE_DATA_FROM_THIS(a) \
  CR (a, DISK_IO_PRIVATE_DATA, DiskIo, DISK_IO_PRIVATE_DATA_SIGNATURE)
```

Example 84. Simple Private Context Data Structure

Example 85 below shows a more complex private context data structure from the IDE *bus driver* from the *EDK*. It contains the signature "ibid" and a `Handle` field that is the child handle for a disk device that the IDE bus driver produced. It also contains pointers to the consumed protocols, which are the Device Path Protocol and PCI I/O Protocol, and storage for the Block I/O Protocol that is produced by this driver. In addition, there are a large number of private data fields that are used during initialization, read operations, write operations, flush operations, and error recovery. Details on how these private fields are used can be found in the source code to the IDE bus driver in the *EDK*.

```
#define IDE_BLK_IO_DEV_SIGNATURE  EFI_SIGNATURE_32 ('i', 'b', 'i', 'd')

typedef struct {
  UINT32                        Signature;

  EFI_HANDLE                    Handle;
  EFI_BLOCK_IO_PROTOCOL         BlkIo;
  EFI_BLOCK_IO_MEDIA            BlkMedia;
  EFI_DISK_INFO_PROTOCOL        DiskInfo;
  EFI_DEVICE_PATH_PROTOCOL      *DevicePath;
  EFI_PCI_IO_PROTOCOL           *PciIo;
  IDE_BUS_DRIVER_PRIVATE_DATA *IdeBusDriverPrivateData;

  //
  // Local Data for IDE interface goes here
  //
  EFI_IDE_CHANNEL               Channel;
  EFI_IDE_DEVICE                Device;
  UINT16                        Lun;
  IDE_DEVICE_TYPE               Type;

  IDE_BASE_REGISTERS            *IoPort;
  UINT16                        AtapiError;

  INQUIRY_DATA                  *pInquiryData;
  EFI_IDENTIFY_DATA             *pIdData;
  ATA_PIO_MODE                  PioMode;
  ATA_UDMA_MODE                 UDma_Mode;
  CHAR8                         ModelName[41];
  REQUEST_SENSE_DATA            *SenseData;
  UINT8                         SenseDataNumber;
  UINT8                         *Cache;

  EFI_UNICODE_STRING_TABLE      *ControllerNameTable;
} IDE_BLK_IO_DEV;

#include "ComponentName.h"

#define IDE_BLOCK_IO_DEV_FROM_THIS(a)           CR (a, IDE_BLK_IO_DEV,
BlkIo, IDE_BLK_IO_DEV_SIGNATURE)
#define IDE_BLOCK_IO_DEV_FROM_DISK_INFO_THIS(a) CR (a, IDE_BLK_IO_DEV,
DiskInfo, IDE_BLK_IO_DEV_SIGNATURE)
```

Example 85. Complex Private Context Data Structure

8.3 Allocating Private Context Data Structures

The private context data structures are allocated in the **Start()** function of the driver's **EFI_DRIVER_BINDING_PROTOCOL**. The service that is typically used to allocate the private context data structures is **gBS->AllocatePool()**. Example 86 below shows the generic template for allocating a private context data structure in the **Start()**function of the **EFI_DRIVER_BINDING_PROTOCOL**. This code example shows only a fragment from the **Start()** function. Chapter 9 covers the services that are produced by the **EFI_DRIVER_BINDING_PROTOCOL** in more detail.

```
EFI_STATUS
EFIAPI
<<DriverName>>DriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )
{
  EFI_STATUS                    Status;
  <<DRIVER_NAME>>_PRIVATE_DATA  Private;

  //
  // Allocate the private context data structure
  //
  Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (<<DRIVER_NAME>>_PRIVATE_DATA),
                (VOID**)&Private
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Clear the contents of the allocated buffer
  //
  gBS->SetMem (Private, sizeof (<<DRIVER_NAME>>_PRIVATE_DATA), 0);
}
```

Example 86. Allocation of a Private Context Data Structure

Example 87 below shows the same generic template for the **Start()** function above except that it uses UEFI driver Library functions to allocate a private context data structure.

```
EFI_STATUS
EFIAPI
<<DriverName>>DriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )
{
  EFI_STATUS                    Status;
  <<DRIVER_NAME>>_PRIVATE_DATA  Private;

  //
  // Allocate and clear the private context data structure
  //
  Private = EfiLibAllocateZeroPool (sizeof
(<<DRIVER_NAME>>_PRIVATE_DATA));
  if (Private == NULL) {
    return EFI_OUT_OF_RESOURCES;
  }
}
```

Example 87. Library Allocation of Private Context Data Structure

Example 88 below shows a code fragment from the disk I/O driver that allocates the private context data structure for the disk I/O driver.

```
EFI_STATUS
EFIAPI
DiskIoDriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )
{
  EFI_STATUS              Status;
  DISK_IO_PRIVATE_DATA    *Private;

  //
  // Initialize the disk I/O device instance.
  //
  Status = gBS->AllocatePool(
                 EfiBootServicesData,
                 sizeof (DISK_IO_PRIVATE_DATA),
                 &Private
                 );
  if (EFI_ERROR (Status)) {
    goto ErrorExit;
  }
  EfiZeroMem (Private, sizeof(DISK_IO_PRIVATE_DATA));
}
```

Example 88. Disk I/O Allocation of Private Context Data Structure

8.4 Freeing Private Context Data Structures

The private context data structures are freed in the **Stop()** function of the driver's **EFI_DRIVER_BINDING_PROTOCOL**. The service that is typically used to free the private context data structures is **gBS->FreePool()**. Example 89 below shows the generic template for allocating a private context data structure in the **Stop()**function of the **EFI_DRIVER_BINDING_PROTOCOL**. This code example shows only a fragment from the **Stop()** service. Chapter 9 covers the services that are produced by the **EFI_DRIVER_BINDING_PROTOCOL** in more detail.

```
EFI_STATUS
EFIAPI
<<DriverName>>DriverBindingStop (
  IN  EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN  EFI_HANDLE                   ControllerHandle,
  IN  UINTN                        NumberOfChildren,
  IN  EFI_HANDLE                   *ChildHandleBuffer
  )
{
  EFI_STATUS                         Status;
  EFI_<<PROTOCOL_NAME_Pm>>_PROTOCOL  *<<ProtocolNamePm>>;
  <<DRIVER_NAME>>_PRIVATE_DATA       Private;

  //
  // Look up one of the driver's produced protocols
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfi<<ProtocolNamePm>>ProtocolGuid,
                &<<ProtocolNamePm>>,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
  if (EFI_ERROR (Status)) {
    return EFI_UNSUPPORTED;
  }

  //
  // Retrieve the private context data structure from the produced
protocol
  //
  Private = <<DRIVER_NAME>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Pm>>_THIS (
              <<ProtocolNamePm>>
              );

  //
  // Free the private context data structure
  //
  Status = gBS->FreePool (Private);
  if (EFI_ERROR (Status)) {
    return Status;
  }
}
```

Example 89. Freeing a Private Context Data Structure

Example 90 below shows code fragments from the disk I/O driver that frees the private context data structure for the disk I/O driver.

```
EFI_STATUS
EFIAPI
DiskIoDriverBindingStop (
  IN  EFI_DRIVER_BINDING_PROTOCOL    *This,
  IN  EFI_HANDLE                     ControllerHandle,
  IN  UINTN                          NumberOfChildren,
  IN  EFI_HANDLE                     *ChildHandleBuffer
  )
{
  EFI_STATUS            Status;
  EFI_DISK_IO_PROTOCOL  *DiskIo;
  DISK_IO_PRIVATE_DATA  *Private;

  //
  // Get our context back.
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiDiskIoProtocolGuid,
                &DiskIo,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
  if (EFI_ERROR (Status)) {
    return EFI_UNSUPPORTED;
  }

  Private = DISK_IO_PRIVATE_DATA_FROM_THIS (DiskIo);

  gBS->FreePool (Private);
}
```

Example 90. Disk I/O Freeing of a Private Context Data Structure

8.5 Protocol Functions

The protocol functions that are produced by an UEFI driver also need to access the private context data structure. These functions typically need access to the consumed protocols and the private data fields to perform the protocol function's required operation. Example 91 below shows a template for the implementation of a protocol function that retrieves the private context data structure using the **CR()** based macro and the *This* pointer for the produced protocol. The **Stop()** function from the **EFI_DRIVER_BINDING_PROTOCOL** uses the same **CR()** based macro to retrieve the private context data structure. The only difference is that the *This* pointer is not passed into the **Stop()** function. Instead, the **Stop()** function uses *ControllerHandle* to retrieve one of the produced protocols and then uses the **CR()** based macro with that protocol interface pointer to retrieve the private context data structure.

```
EFI_STATUS
EFIAPI
<<DriverName>><<ProtocolNamePn>><<FunctionNameM>> (
  IN EFI_<<PROTOCOL_NAME_PN>>_PROTOCOL *This,
  //
  // Additional function arguments here.
  //
  )
{
  <<DRIVER_NAME>>_PRIVATE_DATA  Private;

  //
  // Use This pointer to retrieve the private context structure
  //
  Private = <<DRIVER_NAME>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Pn>>_THIS
(This);
}
```

Example 91. Retrieving the Private Context Data Structure

Example 92 below shows a code fragment from the **ReadDisk()** service of the **EFI_DISK_IO_PROTOCOL** that is produced by the disk I/O driver. It uses the **CR()** macro and the *This* pointer to the **EFI_DISK_IO_PROTOCOL** to retrieve the **DISK_IO_PRIVATE_DATA** private context data structure.

```
EFI_STATUS
EFIAPI
DiskIoReadDisk (
  IN EFI_DISK_IO_PROTOCOL  *This,
  IN UINT32                MediaId,
  IN UINT64                Offset,
  IN UINTN                 BufferSize,
  OUT VOID                 *Buffer
  )
{
  EFI_STATUS             Status;
  DISK_IO_PRIVATE_DATA  *Private;

  Private = DISK_IO_PRIVATE_DATA_FROM_THIS (This);
}
```

Example 92. Retrieving the Disk I/O Private Context Data Structure

9 Driver Binding Protocol

The Driver Binding Protocol provides services that can be used to do the following:

- Connect a driver to a controller.
- Disconnect a driver from a controller.

UEFI drivers that follow the UEFI Driver Model are required to implement the Driver Binding Protocol. This requirement includes the following drivers:

- Device drivers
- Bus drivers
- Hybrid drivers

Root bridge driver, service drivers, and initializing drivers do not produce this protocol.

The Driver Binding Protocol is the most important protocol that a driver produces, because it is the one protocol that is used by the EFI Boot Services **gBS->ConnectController()** and **gBS->DisconnectController()**. These EFI Boot Services are used by the UEFI boot manager to connect the console and boot devices that are required to boot an operating system. The implementation of the Driver Binding Protocol will vary depending upon the driver's class. Chapter 7 describes the various driver classes. The protocol interface structure for the Driver Binding Protocol is listed below for reference.

Protocol Interface Structure

```
typedef struct _EFI_DRIVER_BINDING_PROTOCOL {
  EFI_DRIVER_BINDING_SUPPORTED    Supported;
  EFI_DRIVER_BINDING_START        Start;
  EFI_DRIVER_BINDING_STOP         Stop;
  UINT32                          Version;
  EFI_HANDLE                      ImageHandle;
  EFI_HANDLE                      DriverBindingHandle;
} EFI_DRIVER_BINDING_PROTOCOL;
```

The Driver Binding Protocol contains the following three services:

- **Supported()**
- **Start()**
- **Stop()**

It also contains the following three data fields:

- *Version*
- *ImageHandle*

- *DriverBindingHandle*

The *ImageHandle* and *DriverBindingHandle* fields are typically pre-initialized to **NULL**, and the UEFI driver Library functions automatically fill them in. The *Version* field does need to be initialized by the driver. Higher *Version* values signify a new driver. This field is a 32-bit value, but the values 0x0–0x0f and 0xFFFFfff0–0xFFFFFFFF are reserved for UEFI drivers written by OEMs. IHVs may use the values 0x10–0xFFFFffef. Each time a new version of an UEFI driver is released, the *Version* field should also be increased.

Many drivers use a *Version* value with a xx.xx.xx.xx revision scheme, so the driver can convey minor updates versus major updates. Drivers from third-party vendors should use this value to display the real true *Version* value that they use when referring to this driver. For example, if a driver has a version of 3.0.06, then the correct *Version* value would be 0x00030006. Whatever number scheme is chosen, it must be consistent with previously released drivers. Example 127 below shows how a Driver Binding Protocol is typically declared in a driver.

```
EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
  AbcDriverBindingSupported,    // Supported()
  AbcDriverBindingStart,        // Start()
  AbcDriverBindingStop,         // Stop()
  0x10,                         // Version
  NULL,                         // ImageHandle
  NULL                          // DriverBindingHandle
};
```

Example 93. Driver Binding Protocol Declaration

9.1 Driver Binding Template

The implementation of the Driver Binding Protocol for a specific driver is typically found in the file **<<DriverName>>.c**. This file contains the instance of the **EFI_DRIVER_BINDING_PROTOCOL** along with the implementation of the **Supported()**, **Start()**, and **Stop()** services. The example below shows the template for the Driver Binding Protocol.

```
#define <<DRIVER_NAME>_VERSION 0x00000010

EFI_DRIVER_BINDING_PROTOCOL g<<DriverName>>DriverBinding = {
  <<DriverName>>DriverBindingSupported,  // Supported()
  <<DriverName>>DriverBindingStart,      // Start()
  <<DriverName>>DriverBindingStop,       // Stop()
  <<DRIVER_NAME>>_VERSION,               // Version
  NULL,                                  // ImageHandle
  NULL                                   // DriverBindingHandle
};

EFI_STATUS
<<DriverName>>DriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )
{
}

EFI_STATUS
<<DriverName>>DriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )
{
}

EFI_STATUS
<<DriverName>>DriverBindingStop (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN UINTN                        NumberOfChildren,
  IN EFI_HANDLE                   *ChildHandleBuffer
  )
{
}
```

Example 94. Driver Binding Protocol Template

The **`Supported()`**, **`Start()`**, and **`Stop()`** services are covered in detail in section 9.1 of the *UEFI* Specification, including the algorithms for implementing these services for device drivers and bus drivers. If a driver produces multiple instances of the Driver Binding Protocol, then they will be installed in the driver entry point. Each instance of the Driver Binding Protocol is implemented using the same guidelines. The different instances may share worker functions to reduce the size of the driver.

9.2 Rules for Driver Binding Services

No Driver Binding services are allowed to use any console I/O protocols. If an error condition is detected, the following steps should be followed: 1) save the error to a variable, do not produce any I/O or other protocols for the device, but make sure that you are producing driver diagnostics and driver configuration. 2) Wait for the platform firmware or the user to call into one of those 2 produces protocols and resolve the issue.

9.3 Implementing Supported()

The **Supported()** service performs a quick check to see if a driver supports a controller. If the **Supported()** service passes, then the **Start()** service may be called to ask the driver to bind to a specific controller. During the supported function the state of the device must not be modified at all. The normal way to check whether a device can be supported is to check identifiers such as DeviceID and VendorID and class code for a PCI device. The following example is from the UNDI driver (\Sample\Bus\Pci\Undi\RuntimeDxe\ in *EDK*).

```
  Status = gBS->OpenProtocol (
                  Controller,
                  &gEfiPciIoProtocolGuid,
                  (VOID **) &PciIo,
                  This->DriverBindingHandle,
                  Controller,
                  EFI_OPEN_PROTOCOL_BY_DRIVER
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  Status = PciIo->Pci.Read (
                        PciIo,
                        EfiPciIoWidthUint8,
                        0,
                        sizeof (PCI_CONFIG_HEADER),
                        &Pci
                        );

  if (!EFI_ERROR (Status)) {
    Status = EFI_UNSUPPORTED;

    if (Pci.Hdr.ClassCode[2] == 0x02 && Pci.Hdr.VendorId ==
PCI_VENDOR_ID_INTEL) {
      switch (Pci.Hdr.DeviceId) {
//<snip for DWG space saving>
      case 0x1064:
        Status = EFI_SUCCESS;
      }
    }
  }

  gBS->CloseProtocol (
        Controller,
        &gEfiPciIoProtocolGuid,
        This->DriverBindingHandle,
        Controller
        );

  return Status;
```

Example 95. UNDI Driver Binding Supported

9.4 Implementing Start()

The **Start()** service is called only when the **Supported()** has already been called and returned **EFI_SUCCESS**. This is when any required child handles are created and any required protocols are installed into the handle database. During the **Start()** service

the device is enabled and then any required protocols are produced. The following example is from the AtapiExtPassThru sample driver (\Sample\Bus\Pci\AtapiExtPassThru\Dxe\ in *EDK*).

```
  Status = gBS->OpenProtocol (
                 Controller,
                 &gEfiPciIoProtocolGuid,
                 (VOID **) &PciIo,
                 This->DriverBindingHandle,
                 Controller,
                 EFI_OPEN_PROTOCOL_BY_DRIVER
                 );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  Status = PciIo->Attributes (
                    PciIo,
                    EfiPciIoAttributeOperationEnable,
                    EFI_PCI_IO_ATTRIBUTE_IDE_PRIMARY_IO |
EFI_PCI_IO_ATTRIBUTE_IDE_SECONDARY_IO | EFI_PCI_DEVICE_ENABLE,
                    NULL
                    );
  if (EFI_ERROR (Status)) {
    goto Done;
  }
// install Extended SCSI Passthru
```

Example 96. AtapiPassThru Driver Binding Start

9.5 Implementing Stop()

The **Stop()** service does the opposite of the **Start()** service. It disconnects a driver from a controller and frees any resources that were allocated in the **Start()** services. The services **Start()** and **Stop()** should be in reverse order. This means that the last thing opened/allocated in **Start()** should be the first one closed/de-allocated in **Stop()**. In the following example the protocols are uninstalled first since they were done last in the **Start()** service and then the PCI device is disabled. This demonstrates the opposing order of operations between **Start()** and **Stop()** services.

```
  Status = gBS->OpenProtocol (
                  Controller,
                  &gEfiExtScsiPassThruProtocolGuid,
                  (VOID **) &ScsiPassThru,
                  This->DriverBindingHandle,
                  Controller,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  AtapiScsiPrivate = ATAPI_EXT_SCSI_PASS_THRU_DEV_FROM_THIS
(ScsiPassThru);

  Status = gBS->UninstallProtocolInterface (
                  Controller,
                  &gEfiExtScsiPassThruProtocolGuid,
                  &AtapiScsiPrivate->ScsiPassThru
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  //
  // Release Pci Io protocol on the controller handle.
  //
  AtapiScsiPrivate->PciIo->Attributes (
                             AtapiScsiPrivate->PciIo,
                             EfiPciIoAttributeOperationDisable,
                             EFI_PCI_IO_ATTRIBUTE_IDE_PRIMARY_IO |
EFI_PCI_IO_ATTRIBUTE_IDE_SECONDARY_IO | EFI_PCI_DEVICE_ENABLE,
                             NULL
                             );

  gBS->CloseProtocol (
         Controller,
         &gEfiPciIoProtocolGuid,
         This->DriverBindingHandle,
         Controller
         );

  gBS->FreePool (AtapiScsiPrivate);

  return EFI_SUCCESS;
```

Example 97. AtapiExtPassThru Driver Binding Stop

9.6 Driver Binding Changes for Driver Types

The implementations of the Driver Binding Protocol will change in complexity depending on the driver type. A device driver is fairly simple to implement. A bus driver or a hybrid driver may be more complex because it has to manage both the bus controller and the child controllers. These implementations will be discussed in the following sections.

The **EFI_DRIVER_BINDING_PROTOCOL** is installed onto the driver's image handle. It is possible for a driver to produce more than one instance of the Driver Binding Protocol. All additional instances of the Driver Binding Protocol should be installed onto new handles. The installation of the Driver Binding Protocol is handled by the EFI Library Service **EfiLibInstallAllDriverProtocols()**. This service is covered in more detail

in section 7.1. If an error is generated in the installation of the Driver Binding Protocol, then the entire driver should fail.

9.7 Testing Driver Binding

Once a Driver Binding Protocol is implemented, it can be tested with the following UEFI Shell commands:

- **`load`**
- **`connect`**
- **`disconnect`**
- **`reconnect`**

The UEFI boot manager will also test the services of the Driver Binding Protocol when the drivers for console devices and boot devices are connected so an operating system may be booted.

10 Component Name Protocol

The Component Name Protocol provides a human-readable name for drivers and the devices that drivers manage. This protocol applies only to UEFI drivers that follow the UEFI Driver Model, which includes the following:

- Device drivers
- Bus drivers
- Hybrid drivers

Root bridge drivers, service drivers, and initializing drivers do not produce this protocol. It is recommended that this protocol be implemented for all drivers because it provides an easy method for users to associate the various drivers to the controllers they manage. This protocol is also required by *DIG64.*

The Component Name Protocol provides human-readable names as null-terminated Unicode strings and can support one or more languages. At a minimum, the English language should be supported. The human-readable name that a driver produces is from that specific driver's perspective. If multiple drivers are managing the same controller handle, then there will be multiple instances of the Component Name Protocol that may provide different names for a controller. The consumers of the Component Name Protocol will have to decide which names to display when multiple names are available. For example, a PCI bus driver may produce a name for a PCI slot like "PCI Slot #2," and the driver for the SCSI adapter that is inserted into that same PCI slot may produce a name like "XYZ SCSI Host Controller." Both names describe the same physical device from each driver's perspective, and both names are useful depending on how they are used.

It is also suggested that these human-readable names be limited to about 40 Unicode characters in length so consumers of this protocol can easily fit these strings on standard console devices. The protocol interface structure for the Component Name Protocol is listed below for reference.

Protocol Interface Structure

```
typedef struct _EFI_COMPONENT_NAME_PROTOCOL {
  EFI_COMPONENT_NAME_GET_DRIVER_NAME      GetDriverName;
  EFI_COMPONENT_NAME_GET_CONTROLLER_NAME  GetControllerName;
  CHAR8                                   *SupportedLanguages;
} EFI_COMPONENT_NAME_PROTOCOL;
```

The Component Name Protocol advertises the languages it supports in a data field called *SupportedLanguages*. This data filed is a null-terminated ASCII string that contains one or more ISO 639-2 language codes. Each language code is composed of three ASCII characters. For example, English is specified by "eng," Spanish by "spa," and French by "fra." A consumer of the Component Name Protocol can parse the *SupportedLanguages* data field to see if the protocol supports a language in which the

consumer is interested. This data field can also be used by the implementation of the Component Name Protocol to see if a requested language is supported.

The **GetDriverName()** service is used to retrieve the name of the driver, and the **GetControllerName()** service is used to retrieve the names of the controllers that the driver is currently managing or the names of the child controllers that the driver may have produced. The simplest implementation of the Component Name Protocol provides the name of the driver. The next most complex implementation is for a device driver that provides both the name of the driver and the names of the controllers that the driver is managing. The most complex implementation is for a bus driver or a hybrid driver that produces names for the driver, names for the bus controllers it is managing, and names for the child controllers that the driver has produced. All three of these implementations will be discussed in the sections that follow.

A few UEFI driver Library functions are provided to simplify the implementation of the Component Name Protocol. These library functions aid in registering and retrieving human-readable strings. Some drivers have fixed names for the drivers and controllers that they manage, and other drivers may create names on the fly based on information that is retrieved from the platform or the controller itself. The UEFI driver Library functions of interest are **EfiLibLookupUnicodeString()**, **EfiLibAddUnicodeString()**, and **EfiLibFreeUnicodeStringTable()**.

The **EFI_COMPONENT_NAME_PROTOCOL** must be installed onto the same handle as the **EFI_DRIVER_BINDING_PROTOCOL**. This install operation is done in the driver's entry point. The installation of the **EFI_DRIVER_BINDING_PROTOCOL** and the **EFI_COMPONENT_NAME_PROTOCOL** is covered in section 7.1. If an error is generated in the installation of the Component Name Protocol, then this error should not cause the entire driver to fail.

The implementation of the Component Name Protocol for a specific driver is typically found in the file **ComponentName.c**. This file contains the following:

- The instance of the **EFI_COMPONENT_NAME_PROTOCOL**
- A static table of driver names
- Static tables of controller names
- The implementation of the **GetDriverName()** and **GetControllerName()** services

For drivers that produce dynamic names for controller or children, the allocation and management of these dynamic names will be performed in the **Start()** and **Stop()** services of the **EFI_DRIVER_BINDING_PROTOCOL**. In addition, dynamic name tables require extra fields in the driver's private context data structure that point to the dynamic name tables. See chapter 8 for details on the design of private context data structures.

Once a Component Name Protocol is implemented, it can be tested with the UEFI Shell commands **devices** and **drivers**. Platform vendors are also expected to implement extensions to the UEFI boot manager that allow the user to manage the various devices in a platform. These UEFI boot manager extensions should use the services of the Component Name Protocol to display the names of devices and the drivers that are managing those devices.

10.1 Driver Name

The simplest implementation of the **EFI_COMPONENT_NAME_PROTOCOL** produces a human-readable name for the driver itself and does not provide any names for the controllers or the children that the driver is managing. Example 98 below shows a template for this type of implementation. The instance of the **EFI_COMPONENT_NAME_PROTOCOL** is installed onto the driver handle in the driver's entry point. The **GetControllerName()** service always returns **EFI_UNSUPPORTED**, and the **GetDriverName()** service uses an UEFI driver Library function to retrieve the driver's name in the correct language from a static table of driver names. The static table of driver names contains two elements per entry. The first element is a three-character ASCII string that contains the ISO 639-2 language code for the driver name in the second element. The second element is a Unicode string that represents the name of the driver in the language specified by the first element. The static table is terminated by two **NULL** elements. Example 98 below shows an example with the three supported languages of English, Spanish, and French. Appendix D contains the source code to the disk I/O driver that produces the name of the disk I/O driver only in English.

```
//
// EFI Component Name Protocol
//
EFI_COMPONENT_NAME_PROTOCOL g<<DriverName>>ComponentName = {
  <<DriverName>>ComponentNameGetDriverName,
  <<DriverName>>ComponentNameGetControllerName,
  "engspafra"
};

//
// Static table of driver names in different languages
//
static EFI_UNICODE_STRING_TABLE m<<DriverName>>DriverNameTable[] = {
  { "eng", L"Insert English Driver Name Here" },
  { "spa", L"Insert Spanish Driver Name Here" },
  { "fra", L"Insert French Driver Name Here" },
  { NULL, NULL }
};

EFI_STATUS
<<DriverName>>ComponentNameGetDriverName (
  IN  EFI_COMPONENT_NAME_PROTOCOL  *This,
  IN  CHAR8                        *Language,
  OUT CHAR16                       **DriverName
  )
{
  return EfiLibLookupUnicodeString (
           Language,
           g<<DriverName>>ComponentName.SupportedLanguages,
           m<<DriverName>>DriverNameTable,
           DriverName
           );
}

EFI_STATUS
<<DriverName>>ComponentNameGetControllerName (
  IN  EFI_COMPONENT_NAME_PROTOCOL  *This,
  IN  EFI_HANDLE                   ControllerHandle,
  IN  EFI_HANDLE                   ChildHandle        OPTIONAL,
  IN  CHAR8                        *Language,
  OUT CHAR16                       **ControllerName
  )
{
  return EFI_UNSUPPORTED;
}
```

Example 98. Driver Name Template

10.2 Device Drivers

Device drivers that implement the Component Name Protocol will typically provide the name of the driver using the method described in section 10.1. This means that the implementation of the **GetDriverName()** service is always the same, and the only element that changes is the static table of driver names for each of the supported languages. The **GetControllerName()** service of the Component Name Protocol is where most of the work is performed.

Example 99 below shows the template for the **GetControllerName()** service for a device driver that produces static names for the controllers that it manages. The static controller names in different languages are declared in exactly the same way that the static driver names were declared in section 10.1. The **GetControllerName()** service

needs to make more parameter checks than **GetDriverName()**, because it needs to make sure that the *ControllerHandle* and *ChildHandle* parameters match a controller that the driver is currently managing. Device drivers do not produce any child handles, so the *ChildHandle* parameter must be **NULL**. The *ControllerHandle* parameter must be checked to verify that it is a controller handle that the driver is currently managing. This check is done by opening a protocol on *ControllerHandle* that was opened **BY_DRIVER** in the driver's **Start()** service of the **EFI_DRIVER_BINDING_PROTOCOL**. If the status code returned is **EFI_ALREADY_STARTED**, then *ControllerHandle* is currently managed by the driver. If the status code returned is not **EFI_ALREADY_STARTED**, then *ControllerHandle* is not being managed by the driver.

Once a device driver has verified that *ChildHandle* is **NULL** and *ControllerHandle* represents a controller it is managing, then an UEFI driver Library Service can be used to retrieve the human-readable name of the controller from a static table of controller names in a specified language.

```
//
// Static table of Controller Names in different languages
//
static EFI_UNICODE_STRING_TABLE m<<DriverName>>ControllerNameTable[] = {
  { "eng", L"Insert English Controller Name Here " },
  { "spa", L"Insert Spanish Controller Name Here " },
  { "fra", L"Insert French Controller Name Here " },
  { NULL, NULL }
};

EFI_STATUS
<<DriverName>>ComponentNameGetControllerName (
  IN  EFI_COMPONENT_NAME_PROTOCOL  *This,
  IN  EFI_HANDLE                   ControllerHandle,
  IN  EFI_HANDLE                   ChildHandle        OPTIONAL,
  IN  CHAR8                        *Language,
  OUT CHAR16                       **ControllerName
  )
{
  EFI_STATUS                          Status;
  EFI_<<PROTOCOL_NAME_Cx>>_PROTOCOL  *<<ProtocolNameCx>>;

  //
  // ChildHandle must be NULL for device drivers
  //
  if (ChildHandle != NULL) {
    return EFI_UNSUPPORTED;
  }

  //
  // Make sure this driver is currently managing ControllerHandle
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfi<<ProtocolNameCx>>ProtocolGuid,
                (VOID **)&<<ProtocolNameCx>>,
                g<<DriverName>>DriverBinding.DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );
  if (Status != EFI_ALREADY_STARTED) {
    gBS->CloseProtocol (
           ControllerHandle,
           &gEfi<<ProtocolNameCx>>ProtocolGuid,
           g<<DriverName>>DriverBinding.DriverBindingHandle,
           );
    return EFI_UNSUPPORTED;
  }

  //
  // Look up the controller name from a static table of controller names
  //
  return EfiLibLookupUnicodeString (
           Language,
           g<<DriverName>>ComponentName.SupportedLanguages,
           m<<DriverName>>ControllerNameTable,
           ControllerName
           );
}
```

Example 99. Device Driver with Static Controller Names

Some device drivers are able to extract information from the devices that they manage so they can provide more specific device names. The dynamic generation of controller names is more complex, but it can provide users with the detailed information they require to identify a specific device. For example, a driver for a disk device may be able

to produce a static name such as “Hard Disk,” but a more specific name, such as “Seagate Barracuda ATA ST313620A Hard Disk”, may be much more useful.

To support the dynamic generation of controller names, several additional steps must be taken. The first is that a pointer to the dynamic table of names must be added to the private context data structure for the controllers that a device driver manages. Example 100 below shows the addition of an **EFI_UNICODE_STRING_TABLE** field to the private context data structure discussed in chapter 8.

```
#define <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE
EFI_SIGNATURE_32('A','B','C','D')

typedef struct {
  UINTN                             Signature;
  EFI_HANDLE                        Handle;

  //
  // Pointers to consumed protocols
  //
  EFI_<<PROTOCOL_NAME_C1>>_PROTOCOL  *<<ProtocolNameC1>>;
  EFI_<<PROTOCOL_NAME_C2>>_PROTOCOL  *<<ProtocolNameC2>>;
  // . . .
  EFI_<<PROTOCOL_NAME_Cn>>_PROTOCOL  *<<ProtocolNameCn>>;

  //
  // Produced protocols
  //
  EFI_<<PROTOCOL_NAME_P1>>_PROTOCOL  <<ProtocolNameP1>>;
  EFI_<<PROTOCOL_NAME_P2>>_PROTOCOL  <<ProtocolNameP2>>;
  // . . .
  EFI_<<PROTOCOL_NAME_Pm>>_PROTOCOL  <<ProtocolNamePm>>;

  //
  // Dynamically allocated table of controller names
  //
  EFI_UNICODE_STRING_TABLE           *ControllerNameTable;

  //
  // Additional Private functions and data fields
  //

}<<DRIVER_NAME>>_PRIVATE_DATA;
```

Example 100. Private Context Data Structure with a Dynamic Controller Name Table

The next update is to the **Start()** service of the **EFI_DRIVER_BINDING_PROTOCOL**. The **Start()** service needs to add a controller name in each supported language to *ControllerNameTable* in the private context data structure. The UEFI driver Library function **EfiLibAddUnicodeString()** can be used to add one or more names to a table. The *ContollerNameTable* must be initialized to **NULL** before the first name is added. Example 101 below shows an example of an English name being added to a dynamically allocated table of Unicode names. If more than one language is supported, then there would be an **EfiLibAddUnicodeString()** call for each language. The construction of the Unicode string for each language is not covered here. The format of names stored with devices varies depending on the bus type, and the translation from a bus-specific name format to a Unicode string cannot be standardized.

```
<<DRIVER_NAME>>_PRIVATE_DATA  *Private
CHAR16                        *ControllerName

//
// Get dynamic name from the device being managed
//

//
// Convert the device name to a Unicode string in a supported language
//

//
// Add the device name to the table of names stored in the private
context data
// structure
//
EfiLibAddUnicodeString (
  "eng",
  g<<DriverName>>ComponentName.SupportedLanguages,
  &Private->ControllerNameTable,
  ControllerName
  );
```

Example 101. Adding a Controller Name to a Dynamic Controller Name Table

The **Stop()** service of the **EFI_DRIVER_BINDING_PROTOCOL** also needs to be updated. When a request is made for a driver to stop managing a controller, the table of controller names that were built in the **Start()** service must be freed. The UEFI driver Library function **EfiLibFreeUnicodeStringTable()** can be used to free the table of controller names. Example 102 below shows the code that should be added to the **Stop()** service. The private context data structure will be retrieved by the **Stop()** service so that the private context data structure can be freed. The call to **EfiLibFreeUnicodeStringTable()**should be made just before the private context data structure is freed.

```
<<DRIVER_NAME>>_PRIVATE_DATA  *Private

EfiLibFreeUnicodeStringTable (Private->ControllerNameTable);
```

Example 102. Freeing a Dynamic Controller Name Table

Finally, the **GetControllerName()** service becomes slightly more complex than the static controller name template. Because the table of controller names is now maintained in the private context data structure, the private context data structure needs to be retrieved based on the parameters passed into **GetControllerName()**. This retrieval is achieved by looking up a protocol that the driver has produced on *ControllerHandle* and using a pointer to that protocol and a **CR()** macro to retrieve a pointer to the private context data structure. Then, the private context data structure can be used with the UEFI driver Library function **EfiLibLookupUnicodeString()** to look up the controller's name in the dynamic table of controller names. Example 103 below shows a template for the **GetControllerName()** service that retrieves the controller name from a dynamic table that is stored in the private context data structure.

```
EFI_STATUS
<<DriverName>>ComponentNameGetControllerName (
  IN  EFI_COMPONENT_NAME_PROTOCOL  *This,
  IN  EFI_HANDLE                   ControllerHandle,
  IN  EFI_HANDLE                   ChildHandle        OPTIONAL,
  IN  CHAR8                        *Language,
  OUT CHAR16                       **ControllerName
  )
{
  EFI_STATUS                          Status;
  EFI_<<PROTOCOL_NAME_Cx>>_PROTOCOL  *<<ProtocolNameCx>>;
  EFI_<<PROTOCOL_NAME_Py>>_PROTOCOL  *<<ProtocolNamePy>>;
  <<DRIVER_NAME>>_PRIVATE_DATA       *Private;

  //
  // ChildHandle must be NULL for device drivers
  //
  if (ChildHandle != NULL) {
    return EFI_UNSUPPORTED;
  }

  //
  // Make sure this driver is currently managing ControllerHandle
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfi<<ProtocolNameCx>>ProtocolGuid,
                (VOID **)&<<ProtocolNameCx>>,
                g<<DriverName>>DriverBinding.DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );
  if (Status != EFI_ALREADY_STARTED) {
    gBS->CloseProtocol (
           ControllerHandle,
           &gEfi<<ProtocolNameCx>>ProtocolGuid,
           g<<DriverName>>DriverBinding.DriverBindingHandle,
           );
    return EFI_UNSUPPORTED;
  }

  //
  // Retrieve an instance of a produced protocol from ControllerHandle
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfi<<ProtocolNamePy>>ProtocolGuid,
                (VOID **)&<<ProtocolNamePy>>,
                g<<DriverName>>DriverBinding.DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Retrieve the private context data structure for ControllerHandle
  //
  Private = <<DRIVER_NAME>>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Py>>_THIS (
              <<ProtocolNamePy>>
              );

  //
  // Look up the controller name from a dynamic table of controller names
  //
  return EfiLibLookupUnicodeString (
           Language,
```

```
            g<<DriverName>>ComponentName.SupportedLanguages,
            Private->ControllerNameTable,
            ControllerName
            );
}
```

Example 103. Device Driver with Dynamic Controller Names

10.3 Bus Drivers and Hybrid Drivers

There are many levels of support that a bus driver or hybrid driver may provide for the Component Name Protocol. These drivers can choose to provide a driver name as described in section 10.1. These drivers can also choose to provide names for the bus controllers that they manage and not provide any names for the children that they produce, such as the device drivers described in section 10.2. This section describes what bus drivers and hybrid drivers need to do to provide human-readable names for the child handles that they produce. The human-readable names for child handles can be provided through static or dynamic controller name tables.

Example 104 below shows an example of a driver that uses static name tables for both the bus controller and the child controllers. It first checks to make sure that *ControllerHandle* represents a bus controller that the driver is currently managing. Next it checks to see if *ChildHandle* is **NULL**. If *ChildHandle* is **NULL**, then the caller is asking for the human-readable name for the bus controller that the driver is managing. The **EfiLibLookupUnicodeString()** function is used to look up the human-readable name that is associated with *ControllerHandle*. If *ChildHandle* is not **NULL**, then the caller is requesting the name of a child controller that the driver has produced. First, a test is performed to make sure that *ChildHandle* is a handle that this driver produced. If that test succeeds, then the **EfiLibLookupUnicodeString()** function is used to look up the human-readable name that is associated with *ChildHandle*.

```
//
// Static table of controller names in different languages
//
static EFI_UNICODE_STRING_TABLE m<<DriverName>>ControllerNameTable[] = {
  { "eng", L"Insert English Bus Controller Name Here " },
  { "spa", L"Insert Spanish Bus Controller Name Here " },
  { "fra", L"Insert French Bus Controller Name Here " },
  { NULL, NULL }
};

//
// Static table of child names in different languages
//
static EFI_UNICODE_STRING_TABLE m<<DriverName>>ChildNameTable[] = {
  { "eng", L"Insert English Child Name Here " },
  { "spa", L"Insert Spanish Child Name Here " },
  { "fra", L"Insert French Child Name Here " },
  { NULL, NULL }
};

EFI_STATUS
<<DriverName>>ComponentNameGetControllerName (
  IN  EFI_COMPONENT_NAME_PROTOCOL  *This,
  IN  EFI_HANDLE                   ControllerHandle,
  IN  EFI_HANDLE                   ChildHandle        OPTIONAL,
  IN  CHAR8                        *Language,
  OUT CHAR16                       **ControllerName
  )
{
  EFI_STATUS                            Status;
  EFI_<<PROTOCOL_NAME_Cx>>_PROTOCOL     *<<ProtocolNameCx>>;
  EFI_OPEN_PROTOCOL_INFORMATION_ENTRY   *OpenInfoBuffer;
  UINTN                                 EntryCount;
  UINTN                                 Index;

  //
  // Make sure this driver is currently managing ControllerHandle
  //
  Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfi<<ProtocolNameCx>>ProtocolGuid,
                  (VOID **)&<<ProtocolNameCx>>,
                  g<<DriverName>>DriverBinding.DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_BY_DRIVER
                  );
  if (Status != EFI_ALREADY_STARTED) {
    gBS->CloseProtocol (
           ControllerHandle,
           &gEfi<<ProtocolNameCx>>ProtocolGuid,
           g<<DriverName>>DriverBinding.DriverBindingHandle,
           );
    return EFI_UNSUPPORTED;
  }

  if (ChildHandle == NULL) {
    //
    // Look up the controller name from a static table of controller
names
    //
    return EfiLibLookupUnicodeString (
             Language,
             g<<DriverName>>ComponentName.SupportedLanguages,
             m<<DriverName>>ControllerNameTable,
             ControllerName
             );
  }
```

```
  //
  // Retrieve the list of agents that are consuming one of the protocols
  // on ControllerHandle that the children consume
  //
  Status = gBS->OpenProtocolInformation (
                  ControllerHandle,
                  &gEfi<<ProtocolNameCx>>ProtocolGuid,
                  &OpenInfoBuffer,
                  &EntryCount
                  );
  if (EFI_ERROR (Status)) {
    return EFI_UNSUPPORTED;
  }

  //
  // See if one of the agents is ChildHandle
  //
  Status = EFI_UNSUPPORTED;
  for (Index = 0; Index < EntryCount; Index++) {
    if (OpenInfoBuffer[Index].ControllerHandle == ChildHandle &&

OpenInfoBuffer[Index].Attributes&EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER) {
      Status = EFI_SUCCESS;
    }
  }

  //
  // Free the information buffer
  //
  gBS->FreePool (OpenInfoBuffer);

  //
  // If ChildHandle was not one of the agents, then return
EFI_UNSUPPORTED
  //
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Look up the child name from a static table of child names
  //
  return EfiLibLookupUnicodeString (
           Language,
           g<<DriverName>>ComponentName.SupportedLanguages,
           m<<DriverName>>ChildNameTable,
           ControllerName
           );
}
```

Example 104. Bus Driver with Static Controller and Child Names

The static tables for the controller names and the child names can be substituted with dynamic tables. This substitution requires the private context structure to be updated along with the **`Start()`** and **`Stop()`** services of the **`EFI_DRIVER_BINDING_PROTOCOL`**. Section 10.2 covers how this update is done for the controller names. Example 105 below shows how this update is performed for a bus driver that does not produce names for the bus controllers that it manages but does produce names for the child handles it creates.

```
EFI_STATUS
<<DriverName>>ComponentNameGetControllerName (
  IN  EFI_COMPONENT_NAME_PROTOCOL  *This,
  IN  EFI_HANDLE                   ControllerHandle,
  IN  EFI_HANDLE                   ChildHandle        OPTIONAL,
  IN  CHAR8                        *Language,
  OUT CHAR16                       **ControllerName
  )
{
  EFI_STATUS                            Status;
  EFI_<<PROTOCOL_NAME_Cx>>_PROTOCOL     *<<ProtocolNameCx>>;
  EFI_OPEN_PROTOCOL_INFORMATION_ENTRY   *OpenInfoBuffer;
  UINTN                                 EntryCount;
  UINTN                                 Index;
  EFI_<<PROTOCOL_NAME_Py>>_PROTOCOL     *<<ProtocolNamePy>>;
  <<DRIVER_NAME>>_PRIVATE_DATA          *Private;

  //
  // This version cannot return controller names, so ChildHandle
  // must not be NULL
  //
  if (ChildHandle == NULL) {
    return EFI_UNSUPPORTED;
  }

  //
  // Make sure this driver is currently managing ControllerHandle
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfi<<ProtocolNameCx>>ProtocolGuid,
                (VOID **)&<<ProtocolNameCx>>,
                g<<DriverName>>DriverBinding.DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );
  if (Status != EFI_ALREADY_STARTED) {
    gBS->CloseProtocol (
           ControllerHandle,
           &gEfi<<ProtocolNameCx>>ProtocolGuid,
           g<<DriverName>>DriverBinding.DriverBindingHandle,
           );
    return EFI_UNSUPPORTED;
  }

  //
  // Retrieve the list of agents that are consuming one of the protocols
  // on ControllerHandle that the children consume
  //
  Status = gBS->OpenProtocolInformation (
                ControllerHandle,
                &gEfi<<ProtocolNameCx>>ProtocolGuid,
                &OpenInfoBuffer,
                &EntryCount
                );
  if (EFI_ERROR (Status)) {
    return EFI_UNSUPPORTED;
  }

  //
  // See if one of the agents is ChildHandle
  //
  Status = EFI_UNSUPPORTED;
  for (Index = 0; Index < EntryCount; Index++) {
    if (OpenInfoBuffer[Index].ControllerHandle == ChildHandle &&

OpenInfoBuffer[Index].Attributes&EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER) {
      Status = EFI_SUCCESS;
```

```
    }
  }

  //
  // Free the information buffer
  //
  gBS->FreePool (OpenInfoBuffer);

  //
  // If ChildHandle was not one of the agents, then return
EFI_UNSUPPORTED
  //
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Retrieve the instance of a produced protocol from ChildHandle
  //
  Status = gBS->OpenProtocol (
                 ChildHandle,
                 &gEfi<<ProtocolNamePy>>ProtocolGuid,
                 (VOID **)&<<ProtocolNamePy>>,
                 g<<DriverName>>DriverBinding.DriverBindingHandle,
                 ChildHandle,
                 EFI_OPEN_PROTOCOL_GET_PROTOCOL
                 );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Retrieve the private context data structure for ChildHandle
  //
  Private = <<DRIVER_NAME>>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Py>>_THIS (
              <<ProtocolNamePy>>
              );

  //
  // Look up the controller name from a dynamic table of child controller
names
  //
  return EfiLibLookupUnicodeString (
           Language,
           g<<DriverName>>ComponentName.SupportedLanguages,
           Private->ControllerNameTable,
           ControllerName
           );
}
```

Example 105. Bus Driver with Dynamic Child Names

11 Driver Configuration Protocol

The Driver Configuration Protocol allows users to set the configuration options for the devices that drivers manage. This protocol applies only to UEFI drivers that follow the UEFI Driver Model, which includes the following:

- Device drivers
- Bus drivers
- Hybrid drivers

Root bridge drivers, service drivers, and initializing drivers do not produce this protocol. If a driver requires configuration information, the Driver Configuration Protocol must be implemented. *DIG64* requires that this protocol be implemented in the UEFI drivers that own hardware devices.

The Driver Configuration Protocol provides a user interface for the user to configure devices in one or more languages. At a minimum, the English language should be supported. If multiple drivers are managing the same controller handle, then there may be multiple Driver Configuration Protocols present for that controller handle. The consumers of the Driver Configuration Protocol will have to decide how the multiple drivers that support configuration are presented to the user. For example, a PCI bus driver may produce a mechanism to enable or disable a specific slot, and the driver for the SCSI adapter that is inserted into that same PCI slot may produce configuration options for the SCSI host controller. Both set of configuration options may be useful to the user when managing the platform. The protocol interface structure for the Driver Configuration Protocol is listed below for reference.

Protocol Interface Structure

```
typedef struct _EFI_DRIVER_CONFIGURATION_PROTOCOL {
  EFI_DRIVER_CONFIGURATION_SET_OPTIONS      SetOptions;
  EFI_DRIVER_CONFIGURATION_OPTIONS_VALID    OptionsValid;
  EFI_DRIVER_CONFIGURATION_FORCE_DEFAULTS   ForceDefaults;
  CHAR8                                     *SupportedLanguages;
} EFI_DRIVER_CONFIGURATION_PROTOCOL;
```

The Driver Configuration Protocol advertises the languages it supports in a data field called *SupportedLanguages*. This data filed is a null-terminated ASCII string that contains one or more ISO 639-2 language codes. Each language code is composed of 3 ASCII characters. For example, English is specified by "eng," Spanish by "spa," and French by "fra." A consumer of the Driver Configuration Protocol can parse the *SupportedLanguages* data field to see if the protocol supports a language in which the consumer is interested. The implementation of the Driver Configuration Protocol can also use this data field to see if a requested language is supported.

The **SetOptions()** service uses the console-related protocols such as the Simple Input Protocol and the Simple Text Output Protocol to allow the user to adjust the configuration options for a specific device that is being managed by the driver. This

service is the only way that the driver is allowed to interact with the user through the console device. The driver may provide the typical banner type information here and provide an interactive menu if required. The design of the user interface should target the worst-case console configuration, which is a serial terminal such as VT-100 at 9600 baud. When choosing input keys for the user interface, the worst-case console configuration should be considered. See section 5.16 for suggestions on selecting input keys.

The **`OptionsValid()`** service is used to check if the current set of configuration options are valid for a specific device, and **`ForceDefaults()`** is an optional service that provides a mechanism to place a device in a default configuration.

The implementations of the Driver Configuration Protocol will vary in complexity depending on the driver type. A device driver is fairly simple to implement. A bus driver or a hybrid driver may be more complex because it may have to provide configuration settings for both the bus controller and the child controllers. Both of these implementations will be discussed in the following sections.

The **`EFI_DRIVER_CONFIGURATION_PROTOCOL`** must be installed onto the same handle as the **`EFI_DRIVER_BINDING_PROTOCOL`**. This install operation is done in the driver's entry point. Installing the **`EFI_DRIVER_BINDING_PROTOCOL`** and the **`EFI_DRIVER_CONFIGURATION_PROTOCOL`** is covered in section 7.1. If an error is generated when installing the Driver Configuration Protocol, then this error should not cause the entire driver to fail.

```
The implementation of the Driver Configuration Protocol for a specific
driver is typically found in the file DriverConfiguration.c. This file
contains the instance of the EFI_DRIVER_CONFIGURATION_PROTOCOL along with
the implementation of the SetOptions(), OptionsValid(), and ForceDefault()
services. Example 106 below shows a template for the implementation of
the Driver Configuration Protocol.
EFI_DRIVER_CONFIGURATION_PROTOCOL g<<DriverName>>DriverConfiguration = {
  <<DriverName>>DriverConfigurationSetOptions,
  <<DriverName>>DriverConfigurationOptionsValid,
  <<DriverName>>DriverConfigurationForceDefaults,
  "eng"
};

EFI_STATUS
<<DriverName>>DriverConfigurationSetOptions (
  IN  EFI_DRIVER_CONFIGURATION_PROTOCOL         *This,
  IN  EFI_HANDLE                                ControllerHandle,
  IN  EFI_HANDLE                                ChildHandle  OPTIONAL,
  IN  CHAR8                                     *Language,
  OUT EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED  *ActionRequired
  )
{
}

EFI_STATUS
<<DriverName>>DriverConfigurationOptionsValid (
  IN  EFI_DRIVER_CONFIGURATION_PROTOCOL  *This,
  IN  EFI_HANDLE                         ControllerHandle,
  IN  EFI_HANDLE                         ChildHandle  OPTIONAL
  )
{
}

EFI_STATUS
<<DriverName>>DriverConfigurationForceDefaults (
  IN  EFI_DRIVER_CONFIGURATION_PROTOCOL         *This,
  IN  EFI_HANDLE                                ControllerHandle,
  IN  EFI_HANDLE                                ChildHandle  OPTIONAL,
  IN  UINT32                                    DefaultType,
  OUT EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED  *ActionRequired
  )
{
}
```

Example 106. Driver Configuration Protocol Template

Once a Driver Configuration Protocol is implemented, it can be tested with the UEFI Shell command **drvcfg**. Platform vendors are also expected to implement extensions to the UEFI boot manager that allow the user to configure the various devices in a platform. These UEFI boot manager extensions use the services of the Component Name Protocol to display the names of devices and it will use the services of the Driver Configuration Protocol to allow the user to adjust the configuration settings for each device. The platform vendor also has the ability to validate the configuration of all the devices in the system prior to booting. In addition, the devices can be reset to their default configurations. Forcing default configurations may be automated if the platform firmware detects a corrupt configuration, or the platform vendor may allow the user to select a menu item to force defaults on a specific device or all devices at once.

The Driver Configuration Protocol is available only for devices that a driver is currently managing. Because EFI supports connecting the minimum number of drivers and devices that are required to establish consoles and gain access to the boot device, there may be many configurable devices that cannot be configured. As a result, when the user wishes to enter a "platform configuration" mode, the UEFI boot manager will

be required to connect all drivers to all devices, so the user will be able to see all the configurable devices in the platform.

11.1 Device Drivers

Device drivers will use the template from Example 106. Example 107 below shows the additional code that should be added to be beginning of each of the services of the Driver Configuration Protocol. The first step is to evaluate *ChildHandle*. If it is not **NULL**, then a request is being made to configure a child device. Because device drivers do not produce children, *ChildHandle* must be **NULL**. The next check is to make sure that *ControllerHandle* represents a device that the device driver is currently managing. This check is done by attempting to open a protocol that was opened **BY_DRIVER** in the **Start()** service. If the return code is not **EFI_ALREADY_STARTED**, then *ControllerHandle* is not being managed by this driver, so **EFI_UNSUPPORTED** is returned. The final step is to retrieve a protocol that was produced by the device driver from *ControllerHandle* and then use a **CR()** macro to retrieve the private context data structure for the device being managed.

```
  EFI_STATUS                          Status;
  EFI_<<PROTOCOL_NAME_Cx>>_PROTOCOL   *<<ProtocolNameCx>>;
  EFI_<<PROTOCOL_NAME_Py>>_PROTOCOL   *<<ProtocolNamePy>>;
  <<DRIVER_NAME>>_PRIVATE_DATA        *Private;

  //
  // Child handle must be NULL for a device driver
  //
  if (ChildHandle != NULL) {
    return EFI_UNSUPPORTED;
  }

  //
  // Make sure this driver is currently managing ControllerHandle
  //
  Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfi<<ProtocolNameCx>>ProtocolGuid,
                  (VOID **)&<<ProtocolNameCx>>,
                  g<<DriverName>>DriverBinding.DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_BY_DRIVER
                  );
  if (Status != EFI_ALREADY_STARTED) {
    gBS->CloseProtocol (
           ControllerHandle,
           &gEfi<<ProtocolNameCx>>ProtocolGuid,
           g<<DriverName>>DriverBinding.DriverBindingHandle,
           );
    return EFI_UNSUPPORTED;
  }

  //
  // Retrieve the an instance of a produced protocol from
ControllerHandle
  //
  Status = gBS->OpenProtocol (
                  ControllerHandle,
                  &gEfi<<ProtocolNamePy>>ProtocolGuid,
                  (VOID **)&<<ProtocolNamePy>>,
                  g<<DriverName>>DriverBinding.DriverBindingHandle,
                  ControllerHandle,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Retrieve the private context data structure for ControllerHandle
  //
  Private = <<DRIVER_NAME>>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Py>>_THIS (
              <<ProtocolNamePy>>
              );
```

Example 107. Retrieving the Private Context Data Structure

The implementation of **SetOptions()** will use the Console I/O Services provided by the **EFI_SYSTEM_TABLE** and use **EFI_SIMPLE_TEXT_INPUT_PROTOCOL** and **EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL** to interact with the user. These protocols are described in sections 11.2 and 11.3 of the *UEFI 2.0 Specification*. **SetOptions()** shall not directly access a serial port, keyboard controller, or a VGA controller. The **EFI_SIMPLE_TEXT_INPUT_PROTOCOL** is used to access the console input device that is available from the EFI System Table through **gST->ConIn**. The

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL is used to access the console output device that is available from the EFI System Table through **gST->ConOut**. The specific design of the user interface and the methods that are used to support multiple languages are not covered here. These design decisions are up to the individual driver writer.

SetOptions() stores its configuration information in nonvolatile storage. This configuration information should be stored with the device, so that the configuration information travels with the device if it is moved between platforms. The exact method for retrieving and storing configuration information on a device is device specific and will not be covered here. Typically, drivers will use the services of a bus I/O protocol to access the resources on a device to retrieve and store configuration information. For example, if a PCI controller has a flash device attached to it, the management of that flash device may be exposed through I/O or memory-mapped I/O registers described in the BARs associated with the PCI device. A PCI device driver can use the **Io.Read()**, **Io.Write()**, **Mem.Read()**, or **Mem.Write()** services of the PCI I/O Protocol to access the flash contents to retrieve and store configuration settings. Devices that are integrated onto the motherboard or are part of a FRU may use the UEFI variable Services such as **gRT->GetVariable()** and **gRT->SetVariable()** to store configuration information.

The **OptionsValid()** service will not interact with the user. Instead, it will read the device's current configuration information and make sure that the information contains a valid set of configuration options. If the configuration information cannot be retrieved or if the configuration information appears to be corrupt, then an error is returned.

The **ForceDefault()** service is optional and may simply return **EFI_UNSUPPORTED**. If this service is implemented, then it stores a valid set of default configuration settings in nonvolatile storage.

The implementation of the **SetOptions()** service could use **OptionsValid()** and **ForceDefaults()** to make sure the current options are valid and, if they are not valid, place the controller in its default configuration. The code fragment in Example 108 below shows the code that could be added to **SetOptions()** to implement this feature.

```
EFI_STATUS                                  Status;
EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED  ActionRequired;

Status = This->OptionsValid (This, ControllerHandle, ChildHandle);
if (EFI_ERROR (Status)) {
  Status = This->ForceDefaults (
                  This,
                  ControllerHandle,
                  ChildHandle,
                  EFI_DRIVER_CONFIGURATION_SAFE_DEFAULTS,
                  &ActionRequired
                  );
}
```

Example 108. Validating Options in SetOptions()

11.2 Bus Drivers and Hybrid Drivers

A bus driver or hybrid driver may provide many levels of support for the Driver Configuration Protocol. These drivers can manage configuration settings for the bus

controllers that they manage and not provide any configuration settings for the children that they produce. This choice means that they behave exactly like the device drivers described in section 11.1. This section describes what bus drivers and hybrid drivers need to do if they manage configuration settings for the child handles that they produce.

Example 109 below shows the additional code that should be added to the beginning of each of the services of the Driver Configuration Protocol. The first step is to evaluate *ChildHandle*. If it is **NULL**, then a request is being made to configure the bus controller. If it is not **NULL**, then a request is being made to configure a child. The *ControllerHandle* must always be checked to make sure it represents a device that the driver is currently managing. This check is done by attempting to open a protocol that was opened **BY_DRIVER** in the **Start()** service. If the return code is not **EFI_ALREADY_STARTED**, then *ControllerHandle* is not being managed by this driver, so **EFI_UNSUPPORTED** is returned. If *ChildHandle* is not **NULL**, then an additional check must be made to verify that *ChildHandle* was produced by this driver. If that check passes, then a protocol that was produced on the *ChildHandle* is retrieved and the **CR()** macro is used to retrieve the private context data structure for *ChildHandle*.

```
  EFI_STATUS                              Status;
  EFI_<<PROTOCOL_NAME_Cx>>_PROTOCOL       *<<ProtocolNameCx>>;
  EFI_<<PROTOCOL_NAME_Py>>_PROTOCOL       *<<ProtocolNamePy>>;
  EFI_<<PROTOCOL_NAME_Pz>>_PROTOCOL       *<<ProtocolNamePz>>;
  EFI_OPEN_PROTOCOL_INFORMATION_ENTRY     *OpenInfoBuffer;
  UINTN                                   EntryCount;
  UINTN                                   Index;
  <<DRIVER_NAME>>_PRIVATE_DATA            *Private;

  //
  // Make sure this driver is currently managing ControllerHandle
  //
  Status = gBS->OpenProtocol (
                 ControllerHandle,
                 &gEfi<<ProtocolNameCx>>ProtocolGuid,
                 (VOID **)&<<ProtocolNameCx>>,
                 g<<DriverName>>DriverBinding.DriverBindingHandle,
                 ControllerHandle,
                 EFI_OPEN_PROTOCOL_BY_DRIVER
                 );
  if (Status != EFI_ALREADY_STARTED) {
    gBS->CloseProtocol (
           ControllerHandle,
           &gEfi<<ProtocolNameCx>>ProtocolGuid,
           g<<DriverName>>DriverBinding.DriverBindingHandle,
           );
    return EFI_UNSUPPORTED;
  }

  //
  // If child handle is NULL, then the bus controller is being configured
  //
  if (ChildHandle == NULL) {
    //
    // Retrieve the an instance of a produced protocol from
ControllerHandle
    //
    Status = gBS->OpenProtocol (
                   ControllerHandle,
                   &gEfi<<ProtocolNamePy>>ProtocolGuid,
                   (VOID **)&<<ProtocolNamePy>>,
                   g<<DriverName>>DriverBinding.DriverBindingHandle,
                   ControllerHandle,
                   EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Retrieve the private context data structure for ControllerHandle
    //
    Private = <<DRIVER_NAME>>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Py>>_THIS
(
                <<ProtocolNamePy>>
                );

    //
    // Add user interface code to configure the bus controller here
    //

    return EFI_SUCCESS;
  }

  //
  // Retrieve the list of agents that are consuming one of the protocols
  // on ControllerHandle that the children opened BY_CHILD_CONTROLLER
  //
```

```
  Status = gBS->OpenProtocolInformation (
                 ControllerHandle,
                 &gEfi<<ProtocolNameCx>>ProtocolGuid,
                 &OpenInfoBuffer,
                 &EntryCount
                 );
  if (EFI_ERROR (Status)) {
    return EFI_UNSUPPORTED;
  }

  //
  // See if one of the agents is ChildHandle
  //
  Status = EFI_UNSUPPORTED;
  for (Index = 0; Index < EntryCount; Index++) {
    if (OpenInfoBuffer[Index].ControllerHandle == ChildHandle &&

OpenInfoBuffer[Index].Attributes&EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER) {
      Status = EFI_SUCCESS;
    }
  }

  //
  // Free the information buffer
  //
  gBS->FreePool (OpenInfoBuffer);

  //
  // If ChildHandle was not one of the agents, then return
EFI_UNSUPPORTED
  //
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Retrieve the instance of a produced protocol from ControllerHandle
  //
  Status = gBS->OpenProtocol (
                 ChildHandle,
                 &gEfi<<ProtocolNamePz>>ProtocolGuid,
                 (VOID **)&<<ProtocolNamePz>>,
                 g<<DriverName>>DriverBinding.DriverBindingHandle,
                 ChildHandle,
                 EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Retrieve the private context data structure for ControllerHandle
  //
  Private = <<DRIVER_NAME>>_PRIVATE_DATA_FROM_<<PROTOCOL_NAME_Pz>>_THIS (
              <<ProtocolNamePz>>
              );

  //
  // Add user interface code to configure the child controller here
  //

  return EFI_SUCCESS;
}
```

Example 109. Retrieving the Private Context Data Structure

11.3 Implementing SetOptions() as an Application

One possible design of the Driver Configuration Protocol is to implement the user interface as an UEFI application that is stored with the device or in an EFI system partition. The implementation of the **`SetOptions()`** service would be changed so it does not produce a user interface. It would perform the same parameter checks as before and it would likely still retrieve the private context data structure. Then, instead of producing a user interface, it would use the **`gBS->LoadImage()`** and **`gBS->StartImage()`** services to load and execute the UEFI application that provides the user interface. This application can then either directly update the new configuration settings, or it can pass the new configuration settings back to **`SetOptions()`**.

12 Driver Diagnostics Protocol

The Driver Diagnostics Protocol allows diagnostics to be executed on the devices that drivers manage. This protocol applies only to UEFI drivers that follow the UEFI Driver Model, which includes the following:

- Device drivers
- Bus drivers
- Hybrid drivers

Root bridge drivers, service drivers, and initializing drivers do not produce this protocol. If a driver implements diagnostic services, then the Driver Diagnostics Protocol must be implemented. *DIG64* requires that this protocol be implemented in the UEFI drivers that own hardware devices.

The Driver Diagnostics Protocol provides diagnostics results in one or more languages. At a minimum, the English language should be supported. If multiple drivers are managing the same controller handle, then there may be multiple instances of Driver Diagnostics Protocol present for that controller handle. The consumers of the Driver Diagnostics Protocol will have to decide how the multiple drivers that support diagnostics are presented to the user. For example, a PCI *bus driver* may produce a mechanism to verify the functionality of a specific PCI slot, and the driver for a SCSI adapter that is inserted into that same PCI slot may produce diagnostics for the SCSI host controller. Both sets of diagnostics may be useful to the user when testing the platform. The protocol interface structure for the Driver Diagnostics Protocol is listed below for reference.

Protocol Interface Structure

```
typedef struct _EFI_DRIVER_DIAGNOSTICS_PROTOCOL {
  EFI_DRIVER_DIAGNOSTICS_RUN_DIAGNOSTICS  RunDiagnostics;
  CHAR8                                   *SupportedLanguages;
} EFI_DRIVER_DIAGNOSTICS_PROTOCOL;
```

The Driver Diagnostics Protocol advertises the languages it supports in a data field called *SupportedLanguages*. This data filed is a null-terminated ASCII string that contains one or more ISO 639-2 language codes. Each language code is composed of three ASCII characters. For example, English is specified by "eng," Spanish by "spa," and French by "fra." A consumer of the Driver Diagnostics Protocol can parse the *SupportedLanguages* data field to see if the protocol supports a language in which the consumer is interested. This data field can also be used by the implementation of the Driver Diagnostics Protocol to see if a requested language is supported.

The **RunDiagnostics()** service runs diagnostics on the controller that a driver is managing or a child that the driver has produced. This service is not allowed to use any of the console-I/O-related protocols. Instead, the results of the diagnostics are returned to the caller in a buffer, and the caller may choose to log the results or

display the results of the diagnostics that were executed. The format of the results must be clear and intuitive to the user.

The implementations of the Driver Diagnostics Protocol will change in complexity depending on the driver type. A device driver is fairly simple to implement. A bus driver or a hybrid driver may be more complex because it may provide diagnostics for both the bus controller and the child controllers. These implementations will be discussed in the following sections.

The **EFI_DRIVER_DIAGNOSTICS_PROTOCOL** must be installed on the same handle as the **EFI_DRIVER_BINDING_PROTOCOL**. This install operation is done in the driver's entry point. Installing the **EFI_DRIVER_BINDING_PROTOCOL** and the **EFI_DRIVER_DIAGNOSTICS_PROTOCOL** is covered in section 7.1. If an error is generated when installing the Driver Diagnostics Protocol, then this error should not cause the entire driver to fail.

The implementation of the Driver Diagnostics Protocol for a specific driver is typically found in the file **DriverDiagnostics.c**. This file contains the instance of the **EFI_DRIVER_DIAGNOSTIOCS_PROTOCOL** along with the implementation of **RunDiagnostics()**. Example 110 below shows the template for the Driver Diagnostics Protocol.

```
EFI_DRIVER_DIAGNOSTICS_PROTOCOL g<<DriverName>>DriverDiagnostics = {
  <<DriverName>>DriverDiagnosticsRunDiagnostics,
  "eng"
};

EFI_STATUS
<<DriverName>>DriverDiagnosticsRunDiagnostics (
  IN  EFI_DRIVER_DIAGNOSTICS_PROTOCOL  *This,
  IN  EFI_HANDLE                       ControllerHandle,
  IN  EFI_HANDLE                       ChildHandle  OPTIONAL,
  IN  EFI_DRIVER_DIAGNOSTIC_TYPE       DiagnosticType,
  IN  CHAR8                            *Language,
  OUT EFI_GUID                         **ErrorType,
  OUT UINTN                            *BufferSize,
  OUT CHAR16                           **Buffer
  )
{
}
```

Example 110. Driver Diagnostics Protocol Template

The *DiagnosticType* parameter tells the driver the type of diagnostics to perform. Standard diagnostics must be implemented, and they test basic functionality and should complete in less than 30 seconds. Extended diagnostics are recommended and may take more than 30 seconds to execute. Manufacturing diagnostics are intended to be used in a manufacturing and test environment.

ErrorType, *BufferSize*, and *Buffer* are the return parameters that report the results of the diagnostic. *Buffer* begins with a **NULL**-terminated Unicode string, so the caller of the **RunDiagnostics()** service can display a human-readable diagnostic result. *ErrorType* is a GUID that defines the format of the data buffer that follows the **NULL**-terminated Unicode string. *BufferSize* is the size of *Buffer* that includes the **NULL**-terminated Unicode string and the GUID-specific data buffer. The implementation of **RunDiagnostics()** must allocate *Buffer* using the service **gBS->AllocatePool()**, and it is the caller's responsibility to free this buffer with **gBS->FreePool()**.

Once a Driver Diagnostics Protocol is implemented, it can be tested with the UEFI Shell command **drvdiag**. Platform vendors are also expected to implement extensions to the UEFI boot manager that allow the user to execute diagnostics on the various devices in a platform. These UEFI boot manager extensions use the services of the Component Name Protocol to display the names of devices, and the services of the Driver Diagnostics Protocol are used to execute diagnostics on each device.

The platform vendor also has the ability to execute diagnostics on devices automatically each time the platform is booted. If all the diagnostics are executed, then the boot time may increase. If none of the diagnostics is executed, then there is a chance an operating system may fail to boot due to a failure that could have been detected. The platform vendor will have to make a policy decision regarding diagnostics and may choose to add setup options that allow the user to enable or disable the execution of diagnostics on each boot.

The Driver Diagnostics Protocol is available only for devices that a driver is currently managing. Because EFI supports connecting the minimum number of drivers and devices that are required to establish console and gain access to the boot device, there may be many unconnected devices that support diagnostics. As a result, when the user wishes to enter a "platform configuration" mode, the UEFI boot manager will be required to connect all drivers to all devices, so that the user will be able to see all the devices that support diagnostics.

12.1 Device Drivers

Device drivers that implement the Driver Diagnostics Protocols need to make sure that *ChildHandle* is **NULL** and that *ControllerHandle* represents a device that the driver is currently managing. Example 107 shows the steps required to check these parameters and retrieve the private context data structure. If these checks pass, then the diagnostic will be executed and results will be returned. The diagnostic code will typically use the services of the protocols that the driver produces and the services of the protocols that the driver consumes to verify the operation of the controller. For example, a PCI device driver that consumes the PCI I/O Protocol and produces the Block I/O Protocol can use the services of the PCI I/O Protocol to verify the operation of the PCI controller. The Block I/O Services can be used to verify that the entire driver is working as expected.

12.2 Bus Drivers and Hybrid Drivers

Bus drivers and hybrid drivers that implement the Driver Diagnostics Protocols need to make sure that *ControllerHandle* and *ChildHandle* represent a device that the driver is currently managing. Example 109 shows the steps that are required to check these parameters and retrieve the private context data structure. If these checks pass, then the diagnostic will be executed and the results will be returned. The diagnostic code will typically use the services of the protocols that the driver produces and the services of the protocols that the drivers consumes to verify the operation of the controller. For example, a PCI device driver that consumes the PCI I/O Protocol and produces the Block I/O Protocol can use the services of the PCI I/O Protocol to verify the operation of the PCI controller. The Block I/O Services can be used to verify that the entire driver is working as expected. Bus drivers and hybrid drivers should provide diagnostics for both the bus controller and the child controllers that these types of drivers produce.

Implementing diagnostics for only the bus controller or only the child controllers is strongly discouraged.

12.3 Implementing RunDiagnostics() as an Application

One possible design of the Driver Diagnostics Protocol is to implement the diagnostics as an UEFI application that is stored with the device or in an EFI system partition. The implementation of **`RunDiagnostics()`** would be changed so that it does not directly execute the diagnostics. It would likely perform the same parameter checks as before and it would still retrieve the private context data structure. Then, instead of executing diagnostics, it would use the **`gBS->LoadImage()`** and **`gBS->StartImage()`** services to load and execute the UEFI application that runs the diagnostics. This application would then return the results of the diagnostics back to **`RunDiagnostics()`**.

13 Bus Specific Driver Override Protocol

Some bus drivers are required to produce the Bus Specific Driver Override Protocol. The driver model for a specific bus type must declare if this protocol is required or not. In general, this protocol applies only to bus types that provide containers for drivers on their child devices.

13.1 Producing Bus Specific Driver Override Protocol

At this time, the only bus type that is required to produce this protocol is PCI, and the container for drivers is the PCI option ROM. The PCI bus driver is required to produce the Bus Specific Driver Override Protocol for PCI devices that have an attached PCI option ROM if the PCI option ROM contains one or more loadable UEFI drivers. If a PCI option ROM is not present or the PCI option ROM does not contain any loadable UEFI drivers, than a Bus Specific Driver Override Protocol will not be produced for that PCI device.

13.2 Consuming Bus Specific Driver Override Protocol

The Bus Specific Driver Override Protocol is consumed only by the EFI Boot Service **`ConnectController()`** to determine the order that UEFI drivers are used to attempt to start a device. An UEFI driver never consumes the Bus Specific Driver Override Protocol.

13.3 Implementing Bus Specific Driver Override Protocol

The Bus Specific Driver Override Protocol is simple to implement. It contains one service called **`GetDriver()`** that returns an ordered list of image handles for the UEFI drivers that were loaded from the UEFI driver container. For PCI, the order in which the image handles are returned matches the order in which the UEFI drivers were found in the PCI option ROM, from the lowest address to the highest address. The PCI bus driver is responsible for enumerating the PCI devices on a PCI bus. When a PCI device is discovered, the PCI device is also checked to see if it has an attached PCI option ROM. The PCI option ROM contents must follow the *PCI Specification* (see section1.3.2) for storing one or more images. The PCI bus driver will walk the list of images in a PCI option ROM looking for UEFI drivers. If an UEFI driver is found, it is optionally decompressed using the Decompress Protocol and then loaded, and the driver entry point is called using the EFI Boot Services **`LoadImage()`** and **`StartImage()`**. If **`LoadImage()`** does not return an error, then the UEFI driver must be added to the end of the list of drivers that the Bus Specific Driver Override Protocol for that PCI device

will return when its **GetDriver()** service is called. This addition could be implemented in many ways, including an array of image handles or a linked list of image handles.

13.3.1 Example driver producing Bus Specific Driver Override Protocol

The following group of code examples shows an implementation of the Bus Specific Driver Override Protocol using an array allocated from the pool to manage the list of UEFI driver image handles.

13.3.1.1 Private Context Data Structure Example

Example 111 shows a fragment of the private context structure that is used to manage the child-device-related information in a bus driver. Most private data structures in UEFI drivers should contain a *Signature* field. This field is used in debug builds to make sure that the correct structure is being used by an UEFI driver. If a bad pointer is passed into a function that examines the signature, then an **ASSERT()** will be generated. The signature of "Priv" is just an example. Each UEFI driver should create a new signature value. The *BusSpecificDriverOverride* field is the protocol instance for the Bus Specific Driver Override Protocol. The *NumberOfHandles* field is the number of image handles that the **GetDriver()** function of the Bus Specific Driver Override Protocol can return. The *HandleBufferSize* field is the number of handles that can be stored in the array *HandleBuffer*, and the *HandleBuffer* field is the array of image handles that may be returned by the **GetDriver()** function of the Bus Specific Driver Override Protocol. The **CR()** macro at the bottom of Example 111 will convert the *This* pointer for the Bus Specific Driver Override Protocol to a pointer to the **PRIVATE** structure. This pointer is used by the **GetDriver()** function to retrieve the private context structure.

```
#define PRIVATE_DATA_SIGNATURE   EFI_SIGNATURE_32('P','r','i','v')

typedef struct {
  UINTN                                          Signature;
  . . .
  EFI_PCI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL
BusSpecificDriverOverride;
  UINTN                                          NumberOfHandles;
  UINTN                                          HandleBufferSize;
  EFI_HANDLE                                     *HandleBuffer;
  . . .
} PRIVATE;

#define PRIVATE_FROM_BUS_SPECIFIC_DRIVER_OBVERRIDE_THIS(a) \
  CR(a, PRIVATE, BusSpecificDriverOverride, PRIVATE_DATA_SIGNATURE)
```

Example 111. Private Context for a Bus Specific Driver Override Protocol

13.3.1.2 GetDriver Example

Example 112 is an example implementation of the **GetDriver()** function of the Bus Specific Driver Override Protocol. The first step is to retrieve the private context structure from the *This* pointer. This retrieval is done with the **CR()** macro from Example 111. The next step is to see if there are any image handles in the private structure. If there are none, then **EFI_NOT_FOUND** is returned. The next step is to see if

DriverImageHandle is a pointer to **NULL**. If it is, then the first image handle from *HandleBuffer* is returned. If *DriverImageHandle* is not a pointer to **NULL**, then a search is made through *HandleBuffer* to find a matching handle. If a matching handle is not found, then **EFI_INVALID_PARAMETER** is returned. If a matching handle is found, then the next handle in the array is returned. If the matching handle is the last handle in the array, then **EFI_NOT_FOUND** is returned.

```
EFI_STATUS
GetDriver (
  IN     EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL  *This,
  IN OUT EFI_HANDLE                                 *DriverImageHandle
  )

{
  UINTN    Index;
  PRIVATE  Private;

  Private = PRIVATE_FROM_BUS_SPECIFIC_DRIVER_OVERRIDE_THIS(This);

  if (Private->NumberOfHandles == 0) {
    return EFI_NOT_FOUND;
  }
  if (*DriverImageHandle == NULL) {
    *DriverImageHandle = Private->HandleBuffer[0];
    return EFI_SUCCESS;
  }
  for (Index = 0; Index < Private->NumberOfHandles; Index++) {
    if (*DriverImageHandle == Private->HandleBuffer[Index]) {
      Index = Index + 1;
      if (Index < Private->NumberOfHandles) {
        *DriverImageHandle = Private->HandleBuffer[Index];
        return EFI_SUCCESS;
      } else {
        return EFI_NOT_FOUND;
      }
    }
  }
  return EFI_INVALID_PARAMETER;
}
```

Example 112. GetDriver() Function of a Bus Specific Driver Override Protocol

13.3.1.3 Adding handles to the list example

Example 113 is an example implementation of a worker function that adds the image handle of a driver to the array. This function would be used when the PCI bus driver is scanning the PCI option ROMs for UEFI drivers. As each UEFI driver is loaded, this function would be called to add the image handle of the UEFI driver to the Bus Specific Driver Override Protocol for the PCI controller that is associated with the PCI option ROM.

If there is not enough room in the image handle array, then an array with ten more handles is allocated, the contents of the old array are transferred to the new array, and the old array is freed. If there is not enough memory to allocate the new array, then **EFI_OUT_OF_RESOURCES** is returned. Once there is enough room to store the new image handle, the image handle is added to the end of the array and **EFI_SUCCESS** is returned.

```
EFI_STATUS
AddDriver(
  IN PRIVATE     *Private,
  IN EFI_HANDLE  DriverImageHandle
  )

{
  EFI_STATUS  Status;
  EFI_HANDLE  *NewBuffer;

  if (Private->NumberOfHandles >= Private->HandleBufferSize) {
    Status = gBS->AllocatePool (
                  EfiBootServicesMemory,
                  (Private->HandleBufferSize + 10) * sizeof
(EFI_HANDLE),
                  (VOID **)&NewBuffer
                  );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    gBS->CopyMem (
           NewBuffer,
           Private->HandleBuffer,
           Private->HandleBufferSize * sizeof (EFI_HANDLE)
           );

    Private->HandleBufferSize += 10;
    if (Private->HandleBuffer) {
      gBS->FreePool ( Private->HandleBuffer);
    }
    Private->HandleBuffer = NewBuffer;
  }

  Private->HandleBuffer[Private->NumberOfHandles] = DriverImageHandle;
  Private->NumberOfHandles++;
  return EFI_SUCCESS;
}
```

Example 113. AddDriver() Worker Function

13.3.1.4 Contructing the Private data structure

Example 114 is an example implementation of a constructor function that initializes the fields of the private context structure that are related to this implementation of the Bus Specific Driver Override Protocol. There is only one service in the Bus Specific Driver Override Protocol, so the **`GetDriver()`** service is initialized. In addition, the array of image handles is initialized to contain zero handles. The first call to the **`AddDriver()`** worker function will allocate space for up to ten driver image handles.

```
EFI_STATUS
InitializeBusSpecificDriverOverrideInstance (
  PRIVATE  *Private
  )

{
  Private->BusSpecificDriverOverride.GetDriver = GetDriver;
  Private->NumberOfHandles = 0;
  Private->HandleBuffer = NULL;
  Private->HandleBufferSize = 0;
  return EFI_SUCCESS;
}
```

Example 114. Bus Specific Driver Override Protocol Constructor

14 PCI Driver Design Guidelines

There are several classes of PCI drivers that cooperate to provide support for PCI controllers in a platform. Table 23 lists these PCI drivers.

Table 23. Classes of PCI Drivers

Class of Driver	Description
Root bridge driver	Produces one or more instances of the PCI Root Bridge I/O Protocol.
PCI bus driver	Consumes the PCI Root Bridge I/O Protocol, produces a child handle for each PCI controller, and installs the Device Path Protocol and the PCI I/O Protocol onto each child handle.
PCI driver	Consumes the PCI I/O Protocol and produces an I/O abstraction that provides services for the console devices and boot devices that are required to boot an EFI-compliant operating system.

This chapter will concentrate on the design and implementation of PCI drivers. PCI drivers must follow all of the general design guidelines described in chapter 5. In addition, this chapter covers the guidelines that apply specifically to the management of PCI controllers.

Figure 15 below shows an example PCI driver stack and the protocols that the PCI-related drivers consume and produce. In this example, the platform hardware produces a single PCI root bridge. The PCI Root Bridge I/O Protocol driver accesses the hardware resources to produce a single handle with the **EFI_DEVICE_PATH_PROTOCOL** and the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. The PCI bus driver consumes the services of the **PCI_ROOT_BRIDGE_IO_PROTOCOL**, and uses those services to enumerate the PCI controllers present in the system. In this example, the PCI bus driver detected a disk controller, a graphics controller, and a USB host controller. As a result, the PCI bus driver produces three child handles with the **EFI_DEVICE_PATH_PROTOCOL** and the **EFI_PCI_IO_PROTOCOL**. The driver for the PCI disk controller consumes the services of the **EFI_PCI_IO_PROTOCOL** and produces two child handles with the **EFI_DEVICE_PATH_PROTOCOL** and the **EFI_BLOCK_IO_PROTOCOL**. The PCI driver for the graphics controller consumes the services of the **EFI_PCI_IO_PROTOCOL** and produces the **EFI_GRAPHICS_OUTPUT_PROTOCOL**. Finally, the PCI driver for the USB host controller consumes the services of the **EFI_PCI_IO_PROTOCOL** to produce the **EFI_USB_HOST_CONTROLLER_PROTOCOL**. It is not shown in Figure 15, but the **EFI_USB_HOST_CONTROLLER_PROTOCOL** would then be consumed by the USB bus driver to produce child handles for each USB device, and USB drivers would then manage those child handles. Chapter 15 contains the guidelines for designing USB drivers.

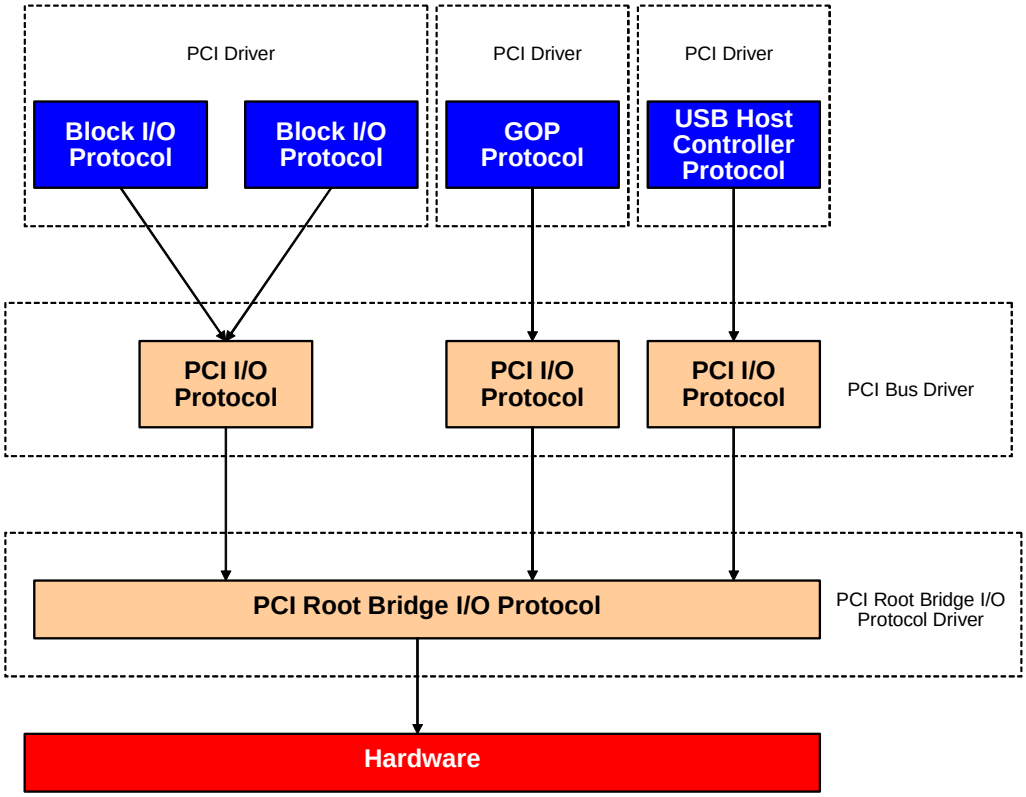

Figure 15. PCI Driver Stack

14.1 PCI Root Bridge I/O Protocol Drivers

An OEM or IBV typically implements the root bridge driver that produces the PCI Root Bridge I/O Protocol. This code is chipset specific and directly accesses the chipset resources to produce the services of the PCI Root Bridge I/O Protocol. A sample driver for systems with a PC-AT-compatible chipset is included in the *EDK*. The source code to this driver is in the directory **\Sample\Chipset\PcCompatible\PciRootBridgeNoEnumeration\Dxe**.

14.2 PCI Bus Drivers

The *EDK* contains a generic PCI bus driver. This driver uses the services of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** to enumerate PCI devices and produce child handle with an **EFI_DEVICE_PATH_PROTOCOL** and an **EFI_PCI_IO_PROTOCOL**. This bus type can support producing one child handle at a time by parsing the *RemainingDevicePath* in its **Supported()** and **Start()** services. However, producing one child handle at a time generally does not make sense because the PCI bus driver

needs to enumerate and assign resources to all of the PCI devices before even a single child handle can be produced. It does not take much extra time to produce the child handles for all the PCI devices that were enumerated, so it is recommended that the PCI bus driver produce all of the PCI devices on the first call to **Start()**.

If EFI-based system firmware is ported to a new platform, most of the PCI-related changes occur in the implementation of the root bridge driver that produces the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Customization of the PCI bus driver is discouraged. As a result, the design and implementation of the PCI bus driver will not be covered in detail.

14.2.1 Hot-Plug PCI Buses

The PCI bus driver in the *EDK* does support hot-plug events in the preboot environment. The PCI bus driver will function correctly with hot-plug-capable hardware and the hot-add, hot-remove, and hot-replace events will be allowed

14.3 PCI Drivers

PCI drivers use the services of the **EFI_PCI_IO_PROTOCOL** to produce one or more protocols that provide I/O abstractions for a PCI controller. PCI drivers follow the UEFI Driver Model, so they may be any of the following:

- Device drivers
- Bus drivers
- Hybrid drivers

The PCI drivers for graphics controllers are typically device drivers that consume the **EFI_PCI_IO_PROTOCOL** and produce the **EFI_GRAPHICS_OUTPUT_PROTOCOL**. The PCI drivers for USB host controllers are typically device drivers that consume the **EFI_PCI_IO_PROTOCOL** and produce the **EFI_USB_HOST_CONTROLLER_PROTOCOL**. The PCI drivers for disk controllers are typically bus drivers or hybrid drivers that consume the **EFI_PCI_IO_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** and produce child handles with the **EFI_DEVICE_PATH_PROTOCOL** and **EFI_BLOCK_IO_PROTOCOL**. PCI drivers for disk controllers that use the SCSI command set will also typically produce the **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** for each SCSI channel that the disk controller produces. Chapter 16 covers the details on SCSI drivers.

14.3.1 Supported()

A PCI driver must implement the **EFI_DRIVER_BINDING_PROTOCOL** that contains the **Supported()**, **Start()**, and **Stop()** services. The **Supported()** service evaluates the *ControllerHandle* that is passed in to see if the *ControllerHandle* represents a PCI device that the PCI driver knows how to manage. The most common method of implementing this test is for the PCI driver to retrieve the PCI configuration header from the PCI controller and evaluate the device ID, vendor ID, and possibly the class code fields of the PCI configuration header. If these fields match the values that the PCI driver knows how to manage, then **Supported()** returns **EFI_SUCCESS**. Otherwise, the **Supported()** service will return **EFI_UNSUPPORTED**. The PCI driver must be careful

not to disturb the state of the PCI controller because a different PCI driver may currently manage the PCI controller.

Example 115 below shows an example of the **Supported()** service for the XYZ PCI driver that manages a PCI controller with a vendor ID of 0x8086 and a device ID of 0xFFFE. First, it attempts to open the PCI I/O Protocol **BY_DRIVER** with **OpenProtocol()**. If the PCI I/O Protocol cannot be opened, then the PCI driver does not support the controller specified by *ControllerHandle*. If the PCI I/O Protocol is opened, then the services of the PCI I/O Protocol are used to read the vendor ID and device ID from the PCI configuration header. The PCI I/O Protocol is always closed with **CloseProtocol()**, and **EFI_SUCCESS** is returned if the vendor ID and device ID match.

```
#define XYZ_VENDOR_ID  0x8086
#define XYZ_DEVICE_ID  0xFFFE

EFI_STATUS
EFIAPI
XyzDriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )
{
  EFI_STATUS           Status;
  EFI_PCI_IO_PROTOCOL  *PciIo;
  UINT16               VendorId;
  UINT16               DeviceId;

  //
  // Open the PCI I/O Protocol on ControllerHandle
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiPciIoProtocolGuid,
                (VOID **) &PciIo,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Read the vendor ID from the PCI configuration header
  //
  Status = PciIo->Pci.Read (
                      PciIo,
                      EfiPciIoWidthUint16,
                      0,
                      sizeof (VendorId),
                      &VendorId
                      );
  if (EFI_ERROR (Status)) {
    goto Done;
  }

  //
  // Read the device ID from the PCI configuration header
  //
  Status = PciIo->Pci.Read (
                      PciIo,
                      EfiPciIoWidthUint16,
                      2,
                      sizeof (DeviceId),
                      &DeviceId
                      );
  if (EFI_ERROR (Status)) {
    goto Done;
  }

  //
  // Evaluate VendorId and DeviceId
  //
  Status = EFI_SUCCESS;
  if (VendorId != XYZ_VENDOR_ID || DeviceId != XYZ_DEVICE_ID) {
    Status = EFI_UNSUPPORTED;
  }

Done:
```

```
  //
  // Close the PCI I/O Protocol
  //

  gBS->CloseProtocol (
         ControllerHandle,
         &gEfiPciIoProtocolGuid,
         This->DriverBindingHandle,
         ControllerHandle
         );

  return Status;
}
```

Example 115. Supported() Service with Partial PCI Configuration Header

The previous example performs two 16-bit reads from the PCI configuration header. The code would be smaller if the entire PCI configuration header was read at once. However, this would increase the execution time because the **Supported()** service would read the entire PCI configuration header for every *ControllerHandle* that was passed in. The **Supported()** service is supposed to be a small, quick check. If a more extensive evaluation of the PCI configuration header is required, then it may make sense to read the entire PCI configuration header at once. Example 116 below shows the same example as above, except it reads the entire PCI configuration header at once in 32-bit chunks.

```
#define XYZ_VENDOR_ID  0x8086
#define XYZ_DEVICE_ID  0xFFFE

EFI_STATUS
EFIAPI
XyzDriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath  OPTIONAL
  )
{
  EFI_STATUS           Status;
  EFI_PCI_IO_PROTOCOL  *PciIo;
  PCI_TYPE00           Pci;

  //
  // Open the PCI I/O Protocol on ControllerHandle
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiPciIoProtocolGuid,
                (VOID **) &PciIo,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Read the entire PCI configuration header
  //
  Status = PciIo->Pci.Read (
                      PciIo,
                      EfiPciIoWidthUint32,
                      0,
                      sizeof (Pci) / sizeof (UINT32),
                      &Pci
                      );
  if (EFI_ERROR (Status)) {
    goto Done;
  }

  //
  // Evaluate VendorId and DeviceId
  //
  Status = EFI_SUCCESS;
  if (Pci.Header.VendorId != XYZ_VENDOR_ID ||
      Pci.Header.DeviceId != XYZ_DEVICE_ID    ) {
    Status = EFI_UNSUPPORTED;
  }

Done:
  //
  // Close the PCI I/O Protocol
  //

  gBS->CloseProtocol (
         ControllerHandle,
         &gEfiPciIoProtocolGuid,
         This->DriverBindingHandle,
         ControllerHandle
         );

  return Status;
}
```

Example 116. Supported() Service with Entire PCI Configuration Header

14.3.2 Start() and Stop()

The **Start()** service of the Driver Binding Protocol for a PCI driver also opens the PCI I/O Protocol **BY_DRIVER**. If the PCI driver is a bus driver or a hybrid driver, then the Device Path Protocol will also be opened **BY_DRIVER**. In addition, all PCI drivers are required to call the **Attributes()** service of the PCI I/O Protocol to enable the I/O, memory, and bus master bits in the Command register of the PCI configuration header. By default, the PCI bus driver is not required to enable the Command register of the PCI controllers. Instead, it is the responsibility of the **Start()** service to enable these bits and of the **Stop()** service to disable these bits.

There is one additional attribute that must be specified in this call to the **Attributes()** service. If the PCI controller is a bus master and capable of generating 64-bit DMA addresses, then the **EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE** attribute must also be enabled. Unfortunately, there is no standard method for detecting if a PCI controller supports 32-bit or 64-bit DMA addresses. As a result, it is the PCI driver's responsibility to inform the PCI bus driver that the PCI controller is capable of producing 64-bit DMA addresses. The PCI bus driver will assume that all PCI controllers are only capable of generating 32-bit DMA addresses unless the PCI driver enables the dual address cycle attribute. The PCI bus driver uses this information along with the services of the PCI Root Bridge I/O Protocol to optimize PCI DMA transactions. If a PCI bus master that is capable of 32-bit DMA addresses is present in a platform that supports more than 4 GB of system memory, then the DMA transactions may have to be double buffered, and this double buffering can reduce the performance of a driver. It is also possible for some platforms to only support system memory above 4 GB. For these reasons, a PCI driver must always accurately describe the DMA capabilities of the PCI controller from the **Start()** service of the Driver Binding Protocol.

Example 117 below shows the code fragment from the **Start()** and **Stop()** services of a PCI driver for a PCI controller that supports 64-bit DMA transactions.

```
EFI_STATUS           Status;
EFI_PCI_IO_PROTOCOL  *PciIo;

//
// Attributes() call from Start() service after the PCI I/O
// Protocol is opened
//
Status = PciIo->Attributes (
                  PciIo,
                  EfiPciIoAttributeOperationEnable,
                  EFI_PCI_DEVICE_ENABLE |
                  EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE,
                  NULL
                  );

//
// Attributes() call from Stop() service before the PCI I/O
// Protocol is closed
//
Status = PciIo->Attributes (
                  PciIo,
                  EfiPciIoAttributeOperationDisable,
                  EFI_PCI_DEVICE_ENABLE |
                  EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE,
                  NULL
                  );
```

Example 117. Start() and Stop() for a 64-Bit DMA Capable PCI Controller

Example 118 below shows the code fragment from the **Start()** and **Stop()** services of a PCI driver for a PCI controller that does not support 64-bit DMA transactions.

```
EFI_STATUS           Status;
EFI_PCI_IO_PROTOCOL  *PciIo;

//
// Attributes() call from Start() service after the PCI I/O
// Protocol is opened
//
Status = PciIo->Attributes (
                  PciIo,
                  EfiPciIoAttributeOperationEnable,
                  EFI_PCI_DEVICE_ENABLE,
                  NULL
                  );

//
// Attributes() call from Stop() service before the PCI I/O
// Protocol is closed
//
Status = PciIo->Attributes (
                  PciIo,
                  EfiPciIoAttributeOperationDisable,
                  EFI_PCI_DEVICE_ENABLE,
                  NULL
                  );
```

Example 118. Start() and Stop() for a 32-Bit DMA Capable PCI Controller

Table 24 lists the **#define** statements that can be used with the **Attributes()** service. A PCI driver must use the **Attributes()** service to enable the decodes on the PCI controller, accurately describe the PCI controller DMA capabilities, and request that specific I/O cycles get forwarded to the device. The call to **Attributes()** will fail if the request cannot be satisfied, and if this failure occurs, then the **Start()** function should

return an error. Once again, any attributes that are enabled in the **Start()** service must be disabled in the **Stop()** service.

Table 24. PCI Attributes

Attribute	Description
EFI_PCI_IO_ATTRIBUTE_ISA_MOTHERBOARD_IO	Used to request the forwarding of I/O cycles 0x0000–0x00FF (10-bit decode).
EFI_PCI_IO_ATTRIBUTE_ISA_IO	Used to request the forwarding of I/O cycles 0x100-0x3FF (10-bit decode).
EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO	Used to request the forwarding of I/O cycles 0x3C6, 0x3C8, and 0x3C9 (10-bit decode).
EFI_PCI_IO_ATTRIBUTE_VGA_MEMORY	Used to request the forwarding of MMIO cycles 0xA0000–0xBFFFF (24-bit decode).
EFI_PCI_IO_ATTRIBUTE_VGA_IO	Used to request the forwarding of I/O cycles 0x3B0–0x3BB and 0x3C0–0x3DF (10-bit decode).
EFI_PCI_IO_ATTRIBUTE_IDE_PRIMARY_IO	Used to request the forwarding of I/O cycles 0x1F0–0x1F7, 0x3F6, 0x3F7 (10-bit decode).
EFI_PCI_IO_ATTRIBUTE_IDE_SECONDARY_IO	Used to request the forwarding of I/O cycles 0x170–0x177, 0x376, 0x377 (10-bit decode).
EFI_PCI_IO_ATTRIBUTE_IO	Enable the I/O decode bit in the Command register.
EFI_PCI_IO_ATTRIBUTE_MEMORY	Enable the Memory decode bit in the Command register.
EFI_PCI_IO_ATTRIBUTE_BUS_MASTER	Enable the Bus Master bit in the Command register.
EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE	Clear for PCI controllers that cannot generate a DAC.
EFI_PCI_IO_ATTRIBUTE_ISA_IO_16	Used to request the forwarding of I/O cycles 0x100–0x3FF (16-bit decode).
EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO_16	Used to request the forwarding of I/O cycles 0x3C6, 0x3C8, and 0x3C9 (16-bit decode).
EFI_PCI_IO_ATTRIBUTE_VGA_IO_16	Used to request the forwarding of I/O cycles 0x3B0–0x3BB and 0x3C0–0x3DF (16-bit decode).

Table 25 lists #define statements are not part of the UEFI 2.0 specification, but are added to the *EDK* for ease of use.

Table 25 EDK Attributes #defines

Attribute	Description
EFI_PCI_DEVICE_ENABLE	Equivalent to combination of **EFI_PCI_IO_ATTRIBUTE_IO**, **EFI_PCI_IO_ATTRIBUTE_MEMORY**, and **EFI_PCI_IO_ATTRIBUTE_BUS_MASTER**.
EFI_VGA_DEVICE_ENABLE	Equivalent to the combination of **EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO**, **EFI_PCI_IO_ATTRIBUTE_VGA_MEMORY**, **EFI_PCI_IO_ATTRIBUTE_VGA_IO**, and **EFI_PCI_IO_ATTRIBUTE_IO**.

Table 26 lists the **#define** statements that can be used with the **GetBarAttributes()** and **SetBarAttributes()** services to adjust the attributes of a memory-mapped I/O region that is described by a Base Address Register (BAR) of a PCI controller. The support of these attributes is optional, and in general, a PCI driver uses these attributes to provide hints that may be used to improve the performance of a PCI driver. This improved performance is especially important for PCI drivers that manage graphics controllers. Any BAR attributes that are set in the **Start()** service must be cleared in the **Stop()** service.

Table 26. PCI BAR Attributes

Attribute	Description
EFI_PCI_IO_ATTRIBUTE_MEMORY_WRITE_COMBINE	Used to map a memory range of a BAR so writes are combined.
EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED	Used to map a memory range of a BAR so all read-write accesses are cached.
EFI_PCI_IO_ATTRIBUTE_MEMORY_DISABLE	Used to disable a memory range of a BAR.

Table 27 lists the **#define** statements that describe some additional attributes of a PCI controller. A PCI driver may retrieve the attributes of a PCI controller with the **Attributes()** service and check to see if these bits are set. The PCI driver may contain different code paths for embedded PCI controllers and PCI controllers that are present on add-in adapters.

Table 27. PCI Embedded Device Attributes

Attribute	Description
EFI_PCI_IO_ATTRIBUTE_EMBEDDED_DEVICE	Clear for an add-in PCI device.
EFI_PCI_IO_ATTRIBUTE_EMBEDDED_ROM	Clear for a physical PCI option ROM accessed through a ROM BAR.

14.3.3 Devices with multiple controllers

Some PCI devices have a series of identical devices on a single device normally behind a PCI bridge. These devices may require additional work as they may need to be controlled by a single instance of the EFI driver. Take the following figure as a sample device.

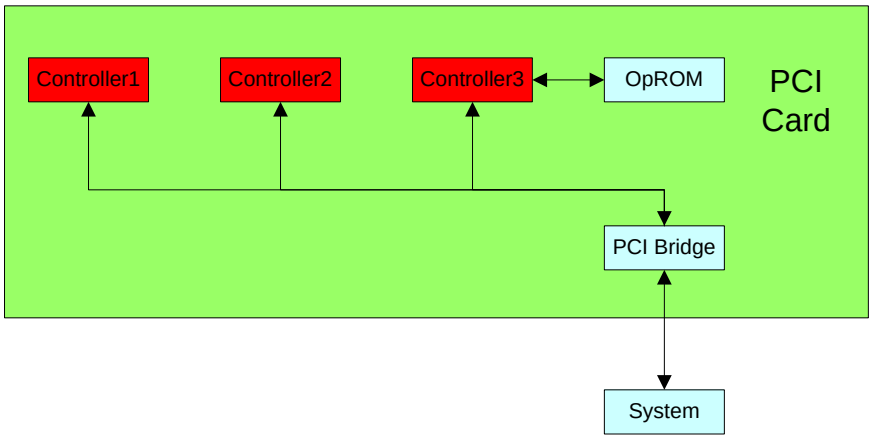

Figure 16. a multi controller PCI device

In this case it may be required for the driver in the OpROM to control all 3 of the controllers on the PCI device. To do this the following actions should be followed:

- In the **`Supported()`** function make sure that you do nothing that actually touches the HW. Very important in this case. If the PCI I/O instance is already opened return the already started error.
- In the **`Start()`** function for Controller1 open the PCI I/O protocol instances on the other handles on the controller using the ByDriver bitmask. This will make all other drivers think that they are already controlled and fail their own **`Supported()`** functions.
    - To scan for other devices on the same bus you can talk to the PCI Root Bridge IO protocol.
    - To scan for other devices across the whole system you can call locateprotocol (for PciIo) and then check for your device non-invasively. Don't modify the HW.
- In the **`Stop()`** function undo everything.

Note: *Note: This can also be used to control several PCI devices with a single controller, but cannot guarantee that the highest version of the driver controls all of them.*

14.4 Accessing PCI Resources

PCI drivers should only access the I/O and memory-mapped I/O resources on the PCI controllers that they manage. They should never attempt to access the I/O or memory-

mapped I/O resource of a PCI controller that they are not managing. They should also never touch the I/O or memory-mapped I/O resources of the chipset or the motherboard.

The PCI I/O Protocol provides services that allow a PCI driver to access easily the resources of the PCI controllers that it is currently managing. These services hide platform-specific implementation details and prevent a PCI driver from inadvertently accessing the resources of the motherboard or other PCI controllers. The PCI I/O Protocol has also been designed to simplify the implementation of PCI drivers. For example, a PCI driver should never read the BARs in the PCI configuration header. Instead, the PCI driver passes in a *BarIndex* and *Offset* into the PCI I/O Protocol services. The PCI bus driver is responsible for managing the PCI controller's BARs.

The services of the PCI I/O Protocol that allow a PCI driver to access the resources on a PCI controller include the following. Chapter 20 contains several examples on how these services should be used.

- **PciIo->PollMem()**
- **PciIo->PollIo()**
- **PciIo->Mem.Read()**
- **PciIo->Mem.Write()**
- **PciIo->Io.Read()**
- **PciIo->Io.Write()**
- **PciIo->Pci.Read()**
- **PciIo->Pci.Write()**
- **PciIo->CopyMem()**

Another important resource that is provided through the PCI I/O Protocol is the PCI option ROM contents. The *RomSize* and *RomImage* fields of the PCI I/O Protocol provide access to a copy of the PCI option ROM contents. These fields may be useful if the PCI driver requires additional information from the contents of the PCI option ROM. It is important to note that the PCI option ROM contents cannot be modified through the *RomImage* field. Modifications to this buffer will only modify the copy of the PCI option ROM contents that are in system memory. The PCI I/O Protocol does not provide services to modify the contents of the actual PCI option ROM.

14.4.1 Memory-Mapped I/O Ordering Issues

PCI transactions follow the ordering rules defined in Appendix E of the *PCI 2.3 Specification.* The ordering rules vary for I/O, memory-mapped I/O, and PCI configuration cycles.

The PCI I/O Protocol **Mem.Read()** service generates PCI memory read cycles that are guaranteed to complete before control is returned to the PCI driver. However, the PCI I/O Protocol **Mem.Write()** service does not guarantee that the PCI memory cycles that are produced by this service will complete before control is returned to the PCI driver. This distinction means that memory write transactions may be sitting in write buffers when this service returns. If the PCI driver requires a **Mem.Write()** transaction to complete, then the **Mem.Write()** transaction must be followed by a **Mem.Read()**

transaction to the same PCI controller. Some chipsets and PCI-to-PCI bridges are more sensitive than others are to this issue. Example 119 below shows a **Mem.Write()** call to a memory-mapped I/O register at offset 0x20 into BAR #1 of a PCI controller. This write transaction is followed by a **Mem.Read()** call from the same memory-mapped I/O register. This combination will guarantee that the write transaction will be completed by the time the **Mem.Read()** call returns.

In general, this mechanism is not required because a PCI driver will typically read a status register, and this read transaction would force all posted write transactions to complete on the PCI controller. The only time this mechanism should be used is when a PCI driver performs a write transaction that is not followed by a read transaction and the PCI driver needs to guarantee that the write transaction is completed immediately.

```
EFI_PCI_IO_PROTOCOL  *PciIo;
UINT32               DmaStartAddress;
UINT16               Word;

//
// Write the value in DmaStartAddress to offset 0x20 of BAR #1
//
PciIo->Mem.Write (
             PciIo,
             EfiPciIoWidthUint32,
             1,
             0x20,
             1,
             &DmaStartAddress
             );

//
// Read offset 0x20 of BAR #1. This guarantees that the previous write
// transaction is posted to the PCI controller.
//
PciIo->Mem.Read (
             PciIo,
             EfiPciIoWidthUint32,
             1,
             0x20,
             1,
             &DmaStartAddress
             );
```

Example 119. Completing a Memory Write Transaction

14.4.2 Hardfail / Softfail

PCI drivers must make sure they do not access resources that are not allocated to any PCI controllers. Doing so may produce unpredictable results including platform hang conditions. For example, if a VGA device is in monochrome mode, accessing the VGA device's color registers may cause unpredictable results. The best rule of thumb here is to access only I/O or memory-mapped I/O resources to which the PCI driver knows for sure the PCI controller will respond. In general, this is not a concern because the PCI I/O Protocol services discussed in section 14.3.3 do not allow the PCI driver to access resources outside the resource ranges described in the BARs of the PCI controllers. However, two mechanisms allow a PCI driver to bypass these safeguards. The first is to use the **EFI_PCI_IO_PASS_THROUGH_BAR** with the PCI I/O Protocol services that provide access to I/O and memory-mapped I/O regions. The second is for a PCI driver to retrieve and use the services of a PCI Root Bridge I/O Protocol.

A PCI driver uses the **EFI_PCI_IO_PASS_THROUGH_BAR** to access ISA resources on a PCI controller. For a PCI driver to use this mechanism safely, the PCI driver must know that the PCI controller it is accessing will actually respond to the I/O or memory-mapped I/O requests in the ISA ranges. The PCI driver can typically know if it will respond by examining the class code, vendor ID, and device ID fields of the PCI controller in the PCI configuration header. The PCI driver must examine the PCI configuration header before any I/O or memory-mapped I/O operations are generated. The PCI configuration header is typically examined in the **Supported()** service, so it is safe to access the ISA resources in the **Start()** service and in the services of the I/O abstraction that the PCI driver is producing. Example 120 below shows an example using the **EFI_PCI_IO_PASS_THROUGH_BAR**.

```
EFI_PCI_IO_PROTOCOL  *PciIo;
UINT8                Data;
UINT16               Word;

//
// Write 0xAA to a Post Card at ISA address 0x80
//
Data = 0xAA;
PciIo->Io.Write(
            PciIo,
            EfiPciIoWidthUint8,
            EFI_PCI_IO_PASS_THROUGH_BAR,
            0x80,
            1,
            &Data
            );

//
// Read the first word from the VGA frame buffer
//
PciIo->Mem.Read(
             PciIo,
             EfiPciIoWidthUint16,
             EFI_PCI_IO_PASS_THROUGH_BAR,
             0xA0000,
             1,
             &Word
             );
```

Example 120. Accessing ISA Resources on a PCI Controller

A PCI driver must also take care when using the services of the PCI Root Bridge I/O Protocol. A PCI driver can retrieve the parent PCI Root Bridge I/O Protocol and use those services to touch any resource on the PCI bus. This touching can be very dangerous because the PCI driver may not know if a different PCI driver owns a resource or not. The use of this mechanism is strongly discouraged and will likely be used only by OEM drivers that have intimate knowledge of the platform and the chipset. Chapter 4 discusses the use of the **gBS->LocateDevicePath()** service, and the example associated with this service shows how the parent PCI Root Bridge I/O Protocol can be retrieved.

Instead of using the parent PCI Root Bridge I/O Protocol, PCI drivers that need to access the resources of other PCI controllers in the platform should search the handle database for controller handles that support the PCI I/O Protocol. To prevent resource conflicts, the PCI I/O Protocols from other PCI controllers should also be opened **BY_DRIVER**. Example 121 below shows how a PCI driver can easily retrieve the list of PCI controller handles in the handle database and use the services of the PCI I/O Protocol on each of those handles to find a peer PCI controller. For example, a PCI adapter that contains multiple PCI controllers behind a PCI-to-PCI bridge may use a

single driver to manage all of the controllers on the adapter. When the PCI driver is connected to the first PCI controller on the adapter, the PCI driver will want to connect to all the other PCI controllers that have the same bus number as the first PCI controller. This example takes advantage of the **`GetLocation()`** service of the PCI I/O Protocol to find matching bus numbers.

```
EFI_STATUS           Status;
UINTN                HandleCount;
EFI_HANDLE           *HandleBuffer;
UINTN                Index;
EFI_PCI_IO_PROTOCOL  *PciIo;
UINTN                MyBus;
UINTN                Seg;
UINTN                Bus;
UINTN                Device;
UINTN                Function;

//
// Retrieve the location of the PCI controller and store the bus
// number in MyBus.
//
Status = PciIo->GetLocation (PciIo, &Seg, &MyBus, &Device, &Function);
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Retrieve the list of handles that support the PCI I/O protocol
// from the handle database. The number of handles that support
// the PCI I/O Protocol is returned in HandleCount, and the array
// of handle values is returned in HandleBuffer.
//
Status = gBS->LocateHandleBuffer (
                ByProtocol,
                &gEfiPciIoProtocolGuid,
                NULL,
                &HandleCount,
                &HandleBuffer
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Loop through all the handles the support the PCI I/O Protocol,
// and retrieve the instance of the PCI I/O Protocol. Use the
// BY_DRIVER open mode, so only PCI I/O Protocols that are not
// currently being managed will be considered.
//
for (Index = 0; Index < HandleCount; Index++) {
  Status = gBS->OpenProtocol (
                 HandleBuffer[Index],
                 &gEfiPciIoProtocolGuid,
                 (VOID **)&PciIo,
                 ImageHandle,
                 NULL,
                 EFI_OPEN_PROTOCOL_BY_DRIVER
                 );
  if (!EFI_ERROR (Status)) {
    //
    // Retrieve the location of the PCI controller and store the
    // bus number in Bus.
    //
    Status = PciIo->GetLocation (PciIo, &Seg, &Bus, &Device, &Function);
    if (!EFI_ERROR (Status)) {
      if (Bus == MyBus) {
        //
        // Store HandleBuffer[Index] so the driver knows it is
        // managing the PCI controller represented by
        // HandleBuffer[Index]. This would typically be stored in
        // the private context data structure
        //

        //
```

```
        // Continue with the next PCI controller in the
        // HandleBuffer array
        //
        continue;
      }
    }
  }

  //
  // Either the handle was already opened by another driver or the
  // bus numbers did not match, so close the PCI I/O Protocol and
  // move on to the next PCI handle.
  //
  gBS->CloseProtocol (
         HandleBuffer[Index],
         &gEfiPciIoProtocolGuid,
         ImageHandle,
         NULL,
         );
}

//
// Free the array of handles that was allocated by
// gBS->LocateHandleBuffer()
//
gBS->FreePool (HandleBuffer);
```

Example 121. Locate PCI Handles with Matching Bus Number

14.4.3 When a PCI Device Does Not Receive Resources

Some PCI controllers may require more resources than the PCI bus can offer. In such cases, the PCI controller must not be visible to PCI drivers because resources were not allocated to the PCI controller. The PCI bus driver will not create a child handle for a PCI controller that does not have allocated resources, and as a result, a PCI driver will never be passed a *ControllerHandle* for a PCI controller that does not have allocated resources. The platform vendor controls the policy decisions that are made when this type of resource-constrained condition is encountered. The PCI driver writer will never have to handle this case.

14.5 PCI DMA

There are three types of DMA transactions that can be implemented using the services of the PCI I/O Protocol:

- Bus master read transactions
- Bus master write transactions
- Common buffer transactions

The PCI I/O Protocol services that are used to manage PCI DMA transactions include the following:

- **PciIo->AllocateBuffer()**
- **PciIo->FreeBuffer()**
- **PciIo->Map()**

- **PciIo->Unmap()**
- **PciIo->Flush()**

14.5.1 Map() Service Cautions

One common mistake is in the use of the **Map()** service. The **Map()** service converts a system memory address to an address that can be used by the PCI device to perform a bus master DMA transaction. The device address that is returned is not related to the original system memory address. Some chipsets maintain a one-to-one mapping between system memory addresses and device addresses on the PCI bus. For this special case, the system memory address and device address will be the same. However, a PCI driver cannot tell if they are executing on a platform with this one-to-one mapping. As a result, a PCI driver must make as few assumptions about the system architecture as possible. Avoiding assumptions means that a PCI driver must never use the device address that is returned from **Map()** to access the contents of the DMA buffer. Instead, this value should only be used to program the base address of the DMA transaction into the PCI controller. This programming is typically accomplished with one or more I/O or memory-mapped I/O write transactions to the PCI controller that the PCI driver is managing.

Example 122 below shows the function prototype for the **Map()** service of the PCI I/O Protocol. A PCI driver can use *HostAddress* to access the contents of the DMA buffer, but the PCI driver should never use the returned parameter *DeviceAddress* to access the contents of the DMA buffer.

```
EFI_STATUS Map (
  IN     EFI_PCI_IO_PROTOCOL             *This,
  IN     EFI_PCI_IO_PROTOCOL_OPERATION   Operation,
  IN     VOID                            *HostAddress,
  IN OUT UINTN                           *NumberOfBytes,
  OUT    EFI_PHYSICAL_ADDRESS            *DeviceAddress,
  OUT    VOID                            **Mapping
  );
```

Example 122. Map() Function

14.5.2 Weakly Ordered Memory Transactions

Some processors, such as the Itanium processor, have weakly ordered memory models. This weak ordering means that system memory transactions may complete in a different order than the source code would seem to indicate. A PCI driver should be implemented so that the source code is compatible with as many processors and platforms as possible. As a result, the guidelines shown here should be followed even if the driver is not initially being written for an Itanium-based platform. The techniques shown here will not have any impact on the executable size of a driver for strongly ordered processors such as IA-32 and EBC.

14.5.3 Bus Master Read / Write Operations

When a DMA transaction is started or stopped, the ownership of the DMA buffer is transitioned from the processor to the DMA bus master and back to the processor. The PCI I/O Protocol provides the **Map()** and **Unmap()** services that are used to set up and complete a DMA transaction. The implementation of the PCI I/O Protocol in the PCI bus

driver uses the **MEMORY_FENCE()** macro to guarantee that all system memory transactions from the processor are completed before the DMA transaction is started. This guarantee prevents the case where a DMA bus master reads from a location in the DMA buffer before a write transaction is flushed from the processor. Because this functionality is built into the PCI I/O Protocol itself, the PCI driver writer does not need to worry about this case for bus master read operations and bus master write operations.

A PCI driver is responsible for flushing all posted write data from a PCI controller when a bus master write operation is completed. First, the PCI driver should read from a register on the PCI controller to guarantee that all the posted write operations are flushed from the PCI controller and through any PCI-to-PCI bridges that are between the PCI controller and the PCI root bridge. Because PCI drivers are polled, they will typically read from a status register on the PCI controller to determine when the bus master write transaction is completed. This read operation is usually sufficient to flush the posted write buffers. The PCI driver must also call the **PciIo->Flush()** service at the end of a bus master write operation. This service flushes all the posted write buffers in the system chipset, so they are guaranteed to be committed to system memory. The combination of the read operation and the **PciIo->Flush()** call guarantee that the bus master's view of system memory and the processor's view of system memory are consistent. Example 123 below shows an example of how a bus master write transaction should be completed and guarantees that the bus master's view of system memory is consistent with the processor's view of system memory.

```
//
// Call PollIo() to poll for Bit #0 in register 0x24 of Bar #1 to
// be set to a 1. This example shows polling a status register to
// wait for a bus master write transaction to complete.
//
Status = PciIo->PollMem (
                  PciIo,
                  EfiPciIoWidthUint32,     // Width
                  1,                       // BarIndex
                  0x24,                    // Offset
                  0x01,                    // Mask
                  0x01,                    // Value
                  10000000,                // Poll for 1 second
                  &Result64                // Result
                  );
if (EFI_ERROR (Status)) {
  return Status;
}

Status = PciIo->Flush (PciIo);
if (EFI_ERROR (Status)) {
  return Status;
}
```

Example 123. Completing a Bus Master Write Operation

14.5.4 Bus Master Common Buffer Operations

Bus master common buffer operations are more complex to manage than bus master read operations and bus master write operations. This complexity is because both the bus master and the processor may simultaneously access a single region of system memory. The memory ordering of PCI transactions generated by the PCI bus master is defined in the *PCI Specification*. However, different processors may use different memory ordering models. As a result, common buffer operations should only be used when they are absolutely required.

If the common buffer memory region can be accessed in a single atomic processor transaction, then no hazards will be present. If the processor has deep write buffers, a write transaction can be delayed, so the **MEMORY_FENCE()** macro can be used to force all processor transactions to complete. If a memory region that the processor needs to read or write requires multiple atomic processor transactions, then hazards may exist if the operations are reordered. If the order that the processor transactions occur is important, then the **MEMORY_FENCE()** macro can be inserted between the processor transactions. However, inserting too many **MEMORY_FENCE()** macros will degrade system performance. For strongly ordered processors, the **MEMORY_FENCE()** macro is a no-op.

A good example where the **MEMORY_FENCE()** macro should be used is when a mailbox data structure is used to communicate between the processor and a bus master. The mailbox typically contains a valid bit that must be set by the processor after the processor has filled the contents of the mailbox. The bus master will scan the mailbox to see if the valid bit it set. When it sees the valid bit, it will read the rest of the mailbox contents and use them to perform an I/O operation. If the processor is weakly ordered, there is a chance that the valid bit will be set before the processor has written all of the other fields in the data structure. To resolve this issue, a **MEMORY_FENCE()** macro should be inserted just before and just after the valid bit is set.

Another mechanism that can be used to resolve these memory-ordering issues is the use of **VOLATILE** variables. If the data structure that is being used as a mailbox is declared in C as **VOLATILE**, then the C compiler will guarantee that all transactions to the **VOLATILE** data structure are strongly ordered. It is recommended that the **MEMORY_FENCE()** macro be used instead of **VOLATILE** data structures.

14.5.5 4 GB Memory Boundary

IA-32 platforms may support more than 4 GB of system memory, but UEFI drivers for IA-32 platforms may only access memory below 4 GB. The 4 GB memory boundary becomes more complex on Itanium-based platforms. Itanium-based platforms do support more than 4 GB of system memory and UEFI drivers running on these platforms can use the memory above and below 4 GB. It is important that UEFI drivers are tested on Itanium-based platforms with both small and large memory configurations to make sure the UEFI driver is not making any assumptions about the system memory configuration.

It is also important to note that some Itanium-based platforms may not map any system memory in the memory region below 4 GB. Instead, all the system memory is mapped above 4 GB. UEFI drivers need to be designed to be compatible with these types of systems too. The general guideline for UEFI drivers is to make as few assumptions about the memory configuration of the platform as possible. This guideline applies to the memory that an UEFI driver allocates and the DMA buffers that a PCI bus master uses.

An UEFI driver should not allocate buffers from specific addresses or below specific addresses. These types of allocations may fail on different system architectures. Likewise, the buffers used for DMA should not be allocated from a specific address or below a specific address. In addition, UEFI drivers should always use the services of the PCI I/O Protocol to set up and complete DMA transactions. It is not legal to program a system memory address into a DMA bus master. This programming will work on chipsets that have a one-to-one mapping between system memory addresses and PCI DMA addresses, but it will not work with chipsets that remap DMA transactions.

The sections that follow contain code examples for the different types of PCI DMA transactions that EFI supports and that show how the PCI I/O Protocol services should be used to maximize the platform compatibility of UEFI drivers.

The *EDK* contains an implementation of the PCI Root Bridge I/O Protocol for a PC-AT-compatible chipset that assumes a one-to-one mapping between system memory address and PCI DMA addresses. It also assumes that DMA operations are not supported above 4 GB. The implementation of the **Map()** and **Unmap()** services in the PCI Root Bridge I/O Protocol handle DMA requests above 4 GB by allocating a buffer below 4 GB and copying the data to the buffer below 4 GB. It is important to realize that these functions will be implemented differently for chipsets that do not assume a one-to-one mapping between system memory addresses and PCI DMA addresses.

14.5.6 DMA Bus Master Read Operation

The general algorithm for performing a bus master read operation is as follows:

- The processor initializes the contents of the DMA using *HostAddress*.
- Call **Map()** with an *Operation* of **EfiPciOperationBusMasterRead**.
- Program the DMA bus master with the *DeviceAddress* returned by **Map()**.
- Program the DMA bus master with the *NumberOfBytes* returned by **Map()**.
- Start the DMA bus master.
- Wait for DMA bus master to complete the bus master read operation.
- Call **Unmap()**.

The code example in Example 124 shows a function for performing a bus master read operation on a PCI controller. The PCI controller is accessed through the parameter *PciIo*. The system memory buffer that will be read by the bus master is specified by *HostAddress* and *Length*. This function performs one or more bus master read operations until either *Length* bytes have been read by the bus master or an error is detected. The PCI controller in this example has three MMIO registers in BAR #1. The MMIO register at offset 0x10 is a status register that the function uses to see if the DMA operation is complete or not. The function writes the start of the DMA transaction to the MMIO register at offset 0x20 and the length of the DMA transaction to the MMIO register at offset 0x24. The write operation to offset 0x24 also starts the DMA read operation. The services of the PCI I/O Protocol that are used in this example include **Map()**, **Unmap()**, **Mem.Write()**, and **PollMem()**. This example is for a 32-bit PCI bus master.

If a 64-bit PCI bus master was being used, then there would be two 32-bit MMIO registers to specify the start address and two 32-bit MMIO registers to specify the length. If the PCI bus master supports 64-bit DMA addressing, then the **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE** attribute must be set in the **Start()** service of the PCI driver.

```
EFI_STATUS
DoBusMasterRead (
  IN EFI_PCI_IO_PROTOCOL  *PciIo,
  IN UINT8                *HostAddress,
  IN UINTN                *Length
  )

{
  EFI_STATUS            Status;
  UINTN                 NumberOfBytes;
  EFI_PHYSICAL_ADDRESS  DeviceAddress;
  VOID                  *Mapping;
  UINT32                DmaStartAddress;
  UINT32                ControllerStatus;

  //
  // Loop until the entire buffer specified by HostAddress and
  // Length has been read from the PCI DMA bus master
  //
  do {
    //
    // Call Map() to retrieve the DeviceAddress to use for the bus
    // master read operation. The Map() function may not support
    // performing a DMA operation for the entire length, so it may
    // be broken up into smaller DMA operations.
    //
    NumberOfBytes = *Length;
    Status = PciIo->Map (
                      PciIo,
                      EfiPciIoOperationBusMasterRead,
                      (VOID *)HostAddress,
                      &NumberOfBytes,
                      &DeviceAddress,
                      &Mapping
                      );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Write the DMA start address to MMIO Register 0x20 of Bar #1
    //
    DmaStartAddress = (UINT32)DeviceAddress;
    Status = PciIo->Mem.Write (
                        PciIo,
                        EfiPciIoWidthUint32,  // Width
                        1,                    // BarIndex
                        0x20,                 // Offset
                        1,                    // Count
                        &DmaStartAddress      // Buffer
                        );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Write the length of the DMA to MMIO Register 0x24 of Bar #1
    // This write operation will also start the DMA transaction
    //
    Status = PciIo->Mem.Write (
                        PciIo,
                        EfiPciIoWidthUint32,  // Width
                        1,                    // BarIndex
                        0x24,                 // Offset
                        1,                    // Count
                        &NumberOfBytes        // Buffer
                        );
    if (EFI_ERROR (Status)) {
```

```
      return Status;
    }

    //
    // Call PollMem() to poll for Bit #0 in MMIO register 0x10 of
    // Bar #1
    //
    Status = PciIo->PollMem (
                      PciIo,
                      EfiPciIoWidthUint32,      // Width
                      1,                        // BarIndex
                      0x10,                     // Offset
                      0x01,                     // Mask
                      0x01,                     // Value
                      10000000,                 // Poll for 1 second
                      &ControllerStatus         // Result
                      );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Call Unmap() to complete the bus master read operation
    //
    Status = PciIo->Unmap (PciIo, Mapping);
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Update the HostAddress and Length remaining based upon the
    // number of bytes transferred
    //
    HostAddress = HostAddress + NumberOfBytes;
    *Length = *Length - NumberOfBytes;
  } while (*Length != 0);

  return Status;
)
```

Example 124. Bus Master Read Operation

14.5.7 DMA Bus Master Write Operation

The general algorithm for performing a bus master write operation is as follows:

- Call **Map()** with an *Operation* of **EfiPciOperationBusMasterWrite**.
- Program the DMA bus master with the *DeviceAddress* returned by **Map()**.
- Program the DMA bus master with the *NumberOfBytes* returned by **Map()**.
- Start the DMA bus master.
- Wait for the DMA bus master to complete the bus master write operation.
- Read any register on the PCI controller to flush all PCI write buffers (see the PCI specification, section 3.2.5.2). In many cases, this read is being done for other purposes, but if not, put in a dummy read.
- Call **Flush()**.
- Call **Unmap()**.

- The processor may read the contents of the DMA buffer using *HostAddress*.

The code example in Example 125 shows a function to perform a bus master write operation on a PCI controller. The PCI controller is accessed through the parameter *PciIo*. The system memory buffer that will be written by the bus master is specified by *HostAddress* and *Length*. This function will perform one or more bus master write operations until either *Length* bytes have been written by the bus master or an error is detected. The PCI controller in this example has three MMIO registers in BAR #1. The MMIO register at offset 0x10 is a status register that the function uses to see if the DMA operation is complete or not. The function writes the start of the DMA transaction to the MMIO register at offset 0x20 and the length of the DMA transaction to the MMIO register at offset 0x24. The write operation to offset 0x24 also starts the DMA write operation. The services of the PCI I/O Protocol that are used in this example include **Map()**, **Unmap()**, **Mem.Write()**, **PollMem()**, and **Flush()**. This example is for a 32-bit PCI bus master. If a 64-bit PCI bus master was being used, then there would be two 32-bit MMIO registers to specify the start address and two 32-bit MMIO registers to specify the length. If the PCI bus master supports 64-bit DMA addressing, then the **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE** attribute must be set in the **Start()** service of the PCI driver.

```
EFI_STATUS
DoBusMasterWrite (
  IN EFI_PCI_IO_PROTOCOL  *PciIo,
  IN UINT8                *HostAddress,
  IN UINTN                *Length
  )

{
  EFI_STATUS            Status;
  UINTN                 NumberOfBytes;
  EFI_PHYSICAL_ADDRESS  DeviceAddress;
  VOID                  *Mapping;
  UINT32                DmaStartAddress;
  UINT32                DummyRead;
  UINT32                ControllerStatus;

  //
  // Loop until the entire buffer specified by HostAddress and
  // Length has been written by the PCI DMA bus master
  //
  do {
    //
    // Call Map() to retrieve the DeviceAddress to use for the bus
    // master write operation. The Map() function may not support
    // performing a DMA operation for the entire length, so it may
    // be broken up into smaller DMA operations.
    //
    NumberOfBytes = *Length;
    Status = PciIo->Map (
                     PciIo,
                     EfiPciIoOperationBusMasterWrite,
                     (VOID *)HostAddress,
                     &NumberOfBytes,
                     &DeviceAddress,
                     &Mapping
                     );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Write the DMA start address to MMIO Register 0x20 of Bar #1
    //
    DmaStartAddress = (UINT32)DeviceAddress;
    Status = PciIo->Mem.Write (
                        PciIo,
                        EfiPciIoWidthUint32,  // Width
                        1,                    // BarIndex
                        0x20,                 // Offset
                        1,                    // Count
                        &DmaStartAddress      // Buffer
                        );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Write the length of the DMA to MMIO Register 0x24 of Bar #1
    // This write operation will also start the DMA transaction
    //
    Status = PciIo->Mem.Write (
                        PciIo,
                        EfiPciIoWidthUint32,  // Width
                        1,                    // BarIndex
                        0x24,                 // Offset
                        1,                    // Count
                        &NumberOfBytes        // Buffer
                        );
```

```
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Call PollMem() to poll for Bit #0 in MMIO register 0x10 of
    // Bar #1
    //
    Status = PciIo->PollMem (
                     PciIo,
                     EfiPciIoWidthUint32,     // Width
                     1,                       // BarIndex
                     0x10,                    // Offset
                     0x01,                    // Mask
                     0x01,                    // Value
                     10000000,                // Poll for 1 second
                     &ControllerStatus        // Result
                     );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Read MMIO Register 0x24 of Bar #1 to flush all posted
    // writes from the PCI bus master and through PCI-to-PCI
    // bridges.
    //
    Status = PciIo->Mem.Read (
                         PciIo,
                         EfiPciIoWidthUint32,  // Width
                         1,                    // BarIndex
                         0x24,                 // Offset
                         1,                    // Count
                         &DummyRead            // Buffer
                         );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Call Flush() to flush all write transactions to system
    // memory
    //
    Status = PciIo->Flush (PciIo);
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Call Unmap() to complete the bus master write operation
    //
    Status = PciIo->Unmap (PciIo, Mapping);
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Update the HostAddress and Length remaining based upon the
    // number of bytes transferred
    //
    HostAddress = HostAddress + NumberOfBytes;
    *Length = *Length - NumberOfBytes;
  } while (*Length != 0);

  return Status;
)
```

Example 125. Bus Master Write Operation

14.5.8 DMA Bus Master Common Buffer Operation

A PCI driver uses common buffers when a memory region requires simultaneous access by both the processor and a PCI bus master. A common buffer is typically allocated in the **`Start()`** service and freed in the **`Stop()`** service. This mechanism is very different from the bus master read and bus master write operations where the PCI driver transfers the ownership of a memory region from the processor to the bus master and back to the processor.

The general algorithm for allocating a common buffer in the **`Start()`** service is as follows:

- Call **`AllocateBuffer()`** to allocate a common buffer.
- Call **`Map()`** with an *`Operation`* of **`EfiPciOperationBusMasterCommonBuffer`**.
- Program the DMA bus master with the *`DeviceAddress`* returned by **`Map()`**.
- The common buffer can now be accessed equally by the processor (using *`HostAddress`*) and the DMA bus master (using *`DeviceAddress`*) .

The general algorithm for freeing a common buffer in the **`Stop()`** service is as follows:

- Call **`Unmap()`**.
- Call **`FreeBuffer()`**.

The code example in Example 126 shows a function that the **Start()** service may call to set up a common buffer operation on a PCI controller. The function accesses the PCI controller through the *PciIo* parameter. The function also allocates a common buffer of *Length* bytes and returns the address of the common buffer in *HostAddress*. A mapping is created for the common buffer and returned in the parameter *Mapping*. The MMIO register at offset 0x18 of BAR #1 is the start address of the common buffer from the PCI controller's perspective. The services of the PCI I/O Protocol that are used in this example include **AllocateBuffer()**, **Map()**, and **Mem.Write()**. This example is for a 32-bit PCI bus master. A 64-bit PCI bus master requires two 32-bit MMIO registers to specify the start address, and the **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE** attribute must be set in the **Start()** service of the PCI driver.

```
EFI_STATUS
SetupCommonBuffer (
  IN EFI_PCI_IO_PROTOCOL  *PciIo,
  IN UINT8                **HostAddress,
  IN UINTN                Length,
  OUT VOID                **Mapping
  )

{
  EFI_STATUS            Status;
  UINTN                 NumberOfBytes;
  EFI_PHYSICAL_ADDRESS  DeviceAddress;
  UINT32                DmaStartAddress;

  //
  // Allocate a common buffer from anywhere in system memory of
  // type EfiBootServicesData.
  //
  Status = PciIo->AllocateBuffer (
                   PciIo,
                   AllocateAnyPages,
                   EfiBootServicesData,
                   EFI_SIZE_TO_PAGES (Length),
                   HostAddress,
                   0
                   );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Call Map() to retrieve the DeviceAddress to use for the bus
  // master common buffer operation. If the Map() function cannot
  // support a DMA operation for the entire length, then return an
  // error.
  //
  NumberOfBytes = Length;
  Status = PciIo->Map (
                   PciIo,
                   EfiPciIoOperationBusMasterCommonBuffer,
                   (VOID *)*HostAddress,
                   &NumberOfBytes,
                   &DeviceAddress,
                   Mapping
                   );
  if (!EFI_ERROR (Status) && NumberOfBytes != Length) {
    PciIo->Unmap (PciIo, *Mapping);
    Status = EFI_OUT_OF_RESOURCES;
  }
  if (EFI_ERROR (Status)) {
    PciIo->FreeBuffer (
             PciIo,
             EFI_SIZE_TO_PAGES (Length),
             (VOID *)*HostAddress
             );
    return Status;
  }

  //
  // Write the DMA start address to MMIO Register 0x18 of Bar #1
  //
  DmaStartAddress = (UINT32)DeviceAddress;
  Status = PciIo->Mem.Write (
                       PciIo,
                       EfiPciIoWidthUint32,  // Width
                       1,                    // BarIndex
                       0x18,                 // Offset
                       1,                    // Count
```

```
                        &DmaStartAddress      // Buffer
                        );

  if (EFI_ERROR (Status)) {
    PciIo->Unmap (PciIo, *Mapping);
    PciIo->FreeBuffer (
              PciIo,
              EFI_SIZE_TO_PAGES (Length),
              (VOID *)*HostAddress
              );
  }

  return Status;
)
```

Example 126. Setting up a Bus Master Common Buffer Operation

The code example in Example 127 shows a function that the **Stop()** service may call to free a common buffer for a PCI controller. The function accesses the PCI controller through the services of the *PciIo* parameter, and the function uses these services to free the common buffer specified by *HostAddress* and *Length*. This function will undo the mapping and free the common buffer. The services of the PCI I/O Protocol that are used in this example include **Unmap()** and **FreeBuffer()**.

```
EFI_STATUS
TearDownCommonBuffer (
  IN EFI_PCI_IO_PROTOCOL  *PciIo,
  IN UINT8                *HostAddress,
  IN UINTN                Length,
  IN VOID                 *Mapping
  )

{
  EFI_STATUS            Status;

  Status = PciIo->Unmap (PciIo, Mapping);
  if (EFI_ERROR (Status)) {
    return Status;
  }

  Status = PciIo->FreeBuffer (
                    PciIo,
                    EFI_SIZE_TO_PAGES (Length),
                    (VOID *)HostAddress
                    );

  return Status;
)
```

Example 127. Tearing Down a Bus Master Common Buffer Operation

15 USB Driver Design Guidelines

There are several classes of USB drivers that cooperate to provide the USB driver stack in a platform. Table 28 lists these USB drivers.

Table 28. Classes of USB Drivers

Class of Driver	Description
Host controller driver	Consumes PCI I/O Protocol on the USB host controller handle and produces the USB2 Host Controller Protocol.
USB bus driver	Consumes the USB2 Host Controller Protocol and produces a child handle for each USB controller on the USB bus. Installs the Device Path Protocol and USB I/O Protocol onto each child handle.
USB device driver	Consumes the USB I/O Protocol and produces an I/O abstraction that provides services for the console devices and boot devices that are required to boot an EFI-compliant operating system.

This chapter will concentrate on how to write host controller drivers and USB device drivers. USB drivers must follow all of the general design guidelines described in chapter 5. In addition, any USB host controllers that are PCI controllers must also follow the PCI-specific design guidelines (see chapter 14).

Figure 17 below shows an example of a USB driver stack and the protocols that the USB drivers consume and produce. Because the USB hub is a special kind of device that simply acts as a signal repeater, it is not included in Figure 17. The protocol interfaces for the USB2 Host Controller Protocol and the EFI USB I/O Protocol are listed below.

Protocol Interface Structure

```
typedef struct _EFI_USB2_HC_PROTOCOL {
EFI_USB2_HC_PROTOCOL_GET_CAPABILITY                 GetCapability;
EFI_USB2_HC_PROTOCOL_RESET                          Reset;
EFI_USB2_HC_PROTOCOL_GET_STATE                      GetState;
EFI_USB2_HC_PROTOCOL_SET_STATE                      SetState;
EFI_USB2_HC_PROTOCOL_CONTROL_TRANSFER               ControlTransfer;
EFI_USB2_HC_PROTOCOL_BULK_TRANSFER                  BulkTransfer;
EFI_USB2_HC_PROTOCOL_ASYNC_INTERRUPT_TRANSFER
                    AsyncInterruptTransfer;
EFI_USB2_HC_PROTOCOL_SYNC_INTERRUPT_TRANSFER
                    SyncInterruptTransfer;
EFI_USB2_HC_PROTOCOL_ISOCHRONOUS_TRANSFER           IsochronousTransfer;
EFI_USB2_HC_PROTOCOL_ASYNC_ISOCHRONOUS_TRANSFER
                    AsyncIsochronousTransfer;
EFI_USB2_HC_PROTOCOL_GET_ROOTHUB_PORT_STATUS        GetRootHubPortStatus;
EFI_USB2_HC_PROTOCOL_SET_ROOTHUB_PORT_FEATURE
                    SetRootHubPortFeature;
EFI_USB2_HC_PROTOCOL_CLEAR_ROOTHUB_PORT_FEATURE
                    ClearRootHubPortFeature;
UINT16                                              MajorRevision;
UINT16                                              MinorRevision;
} EFI_USB2_HC_PROTOCOL;

typedef struct _EFI_USB_IO_PROTOCOL {
EFI_USB_IO_CONTROL_TRANSFER             UsbControlTransfer;
EFI_USB_IO_BULK_TRANSFER                UsbBulkTransfer;
EFI_USB_IO_ASYNC_INTERRUPT_TRANSFER     UsbAsyncInterruptTransfer;
EFI_USB_IO_SYNC_INTERRPUT_TRANSFER      UsbSyncInterruptTransfer
EFI_USB_IO_ISOCHRONOUS_TRANSFER         UsbIsochronousTransfer;
EFI_USB_IO_ASYNC_ISOCHRONOUS_TRANSFER   UsbAsyncIsochronousTransfer;
EFI_USB_IO_GET_DEVICE_DESCRIPTOR        UsbGetDeviceDescriptor;
EFI_USB_IO_GET_CONFIG_DESCRIPTOR        UsbGetConfigDescriptor;
EFI_USB_IO_GET_INTERFACE_DESCRIPTOR     UsbGetInterfaceDescriptor;
EFI_USB_IO_GET_ENDPOINT_DESCRIPTOR      UsbGetEndpointDescriptor;
EFI_USB_IO_GET_STRING_DESCRIPTOR        UsbGetStringDescriptor;
EFI_USB_IO_GET_SUPPORTED_LANGUAGES      UsbGetSupportedLanguages;
EFI_USB_IO_PORT_RESET                   UsbPortReset;
} EFI_USB_IO_PROTOCOL;
```

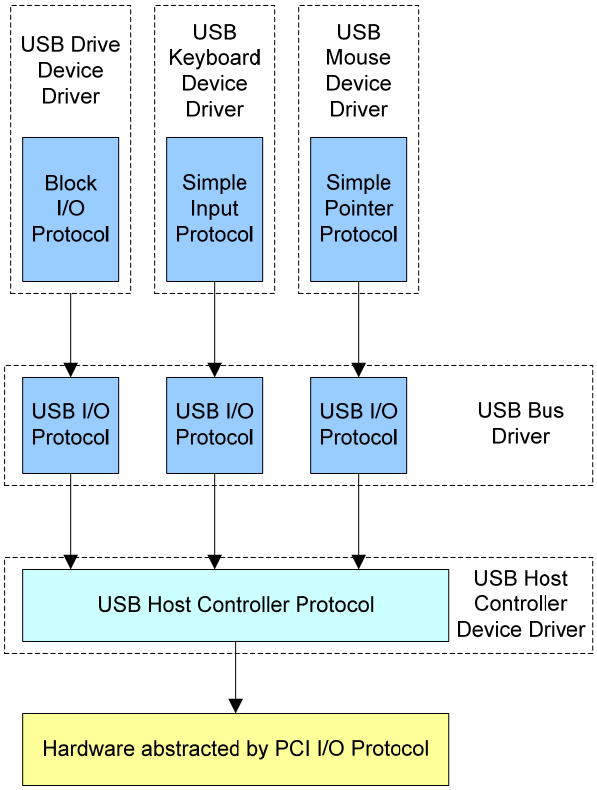

Figure 17. USB Driver Stack

In this example, the platform hardware produces a single USB host controller on the PCI bus. The PCI bus driver will produce a handle with **EFI_DEVICE_PATH_PROTOCOL** and **EFI_PCI_IO_PROTOCOL** installed for this USB host controller. The USB host controller driver will then consume **EFI_PCI_IO_PROTOCOL** on that USB host controller device handle and install the **EFI_USB2_HC_PROTOCOL** onto the same handle.

The USB bus driver consumes the services of **EFI_USB2_HC_PROTOCOL**. It uses these services to enumerate the USB bus. In this example, the USB bus driver detected a USB keyboard, a USB mouse, and two USB mass storage devices. As a result, the USB bus driver will create four child handles and will install the **EFI_DEVICE_PATH_PROTOCOL** and **EFI_USB_IO_PROTOCOL** onto each of those handles.

The USB mouse driver will consume the **EFI_USB_IO_PROTOCOL** and produce the **EFI_SIMPLE_POINTER_PROTOCOL**. The USB keyboard driver will consume the **EFI_USB_IO_PROTOCOL** to produce the **EFI_SIMPLE_TEXT_INPUT_PROTOCOL**. The USB mass storage driver will consume the **EFI_USB_IO_PROTOCOL** to produce the **EFI_BLOCK_IO_PROTOCOL**.

15.1 USB Host Controller Driver

The USB host controller driver depends on which USB host controller specification that the host controller is based. Currently, the major types of USB host controllers are the following:

- Universal Host Controller Interface (UHCI) (USB 1.0 and USB 1.1)
- Open Host Controller Interface (OHCI) (USB 1.0 and USB 1.1)
- Enhanced Host Controller Interface (EHCI) (USB 2.0)

The USB host controller driver is a device driver, which follows the UEFI Driver Model. It typically consumes the services of **EFI_PCI_IO_PROTOCOL** and produces **EFI_USB2_HC_PROTOCOL**. The following section provides guidelines for implementing the **EFI_DRIVER_BINDING_PROTOCOL** services and **EFI_USB2_HC_PROTOCOL** services for the USB host controller driver.

15.1.1 Sample USB host controller drivers in the EDK

- UHCI host controller driver is in the directory **\Sample\Bus\Pci\Uhci\Dxe**.
- EHCI host controller driver is in the directory **\Sample\Bus\Pci\Ehci\Dxe.**

15.1.2 Driver Binding Protocol

The USB host controller driver must install (in its driver entry point function) the Driver Binding protocol. The following sections give guidelines for implementing each of the 3 functions.

15.1.2.1 Supported()

The USB host controller driver must implement the **EFI_DRIVER_BINDING_PROTOCOL** that contains the **Supported()**, **Start()**, and **Stop()** services. The **Supported()** service evaluates the *ControllerHandle* that is passed in to check if the *ControllerHandle* represents a USB host controller that the USB host controller driver knows how to manage. The typical method of implementing this evaluation is for the USB host controller driver to retrieve the PCI configuration header from this controller and check the Class Code field and possibly other fields such as the Device ID and Vendor ID. If all these fields match the values that the USB host controller driver knows how to manage, then the **Supported()** service will return **EFI_SUCCESS**. Otherwise, the **Supported()** service will return **EFI_UNSUPPORTED**.

Example 128 below shows an example of the **Supported()** service for the USB host controller driver that manages a PCI controller with Class code 0x30c. First, it attempts to open the PCI I/O Protocol **BY_DRIVER**. If the PCI I/O Protocol cannot be opened, then the USB host controller driver does not support the controller that is specified by *ControllerHandle*. If the PCI I/O Protocol is opened, then the services of the PCI I/O Protocol are used to read the Class Code from the PCI configuration header. The PCI I/O Protocol is always closed with **CloseProtocol()**, and **EFI_SUCCESS** is returned if the Class Code field match.

```
//
// USB Base Class Code, Subclass Code, and Programming Interface.
//
#define PCI_CLASSC_BASE_CLASS_SERIAL              0x0c
#define PCI_CLASSC_SUBCLASS_SERIAL_USB            0x03
#define PCI_CLASSC_PI_UHCI                        0x00

//
// Class Code Register offset in PCI configuration space
//
#define CLASSC                       0x09


EFI_STATUS
UHCIDriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL                    *This,
  IN EFI_HANDLE                                     Controller,
  IN EFI_DEVICE_PATH_PROTOCOL                       *RemainingDevicePath
  )
{
  EFI_STATUS                                  Status;
  EFI_PCI_IO_PROTOCOL                         *PciIo;
  UINT8                                       UsbClassCReg[3];

  //
  // Test whether there is PCI IO Protocol attached on the
  // controller handle.
  //
  Status = gBS->OpenProtocol (
                         Controller,
                         &gEfiPciIoProtocolGuid,
                         &PciIo,
                         This->DriverBindingHandle,
                         Controller,
                         EFI_OPEN_PROTOCOL_BY_DRIVER
                         );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  Status = PciIo->Pci.Read (
                          PciIo,
                          EfiPciIoWidthUint8,
                          CLASSC,
                          3 *  sizeof(UINT8),
                          &UsbClassCReg
                          );

  if (EFI_ERROR(Status)) {

      Status = EFI_UNSUPPORTED;
      goto Exit;
  }

  //
  // Test whether the controller belongs to UHCI type
  //
  if ((UsbClassCReg[2] != PCI_CLASSC_BASE_CLASS_SERIAL)
        || (UsbClassCReg[1] != PCI_CLASSC_SUBCLASS_SERIAL_USB)
        || (UsbClassCReg.[0] != PCI_CLASSC_PI_UHCI)) {

      Status = EFI_UNSUPPORTED;
      goto Exit;
  }


Exit:
  gBS->CloseProtocol (
```

```
          Controller,
          &gEfiPciIoProtocolGuid,
          This->DriverBindingHandle,
          Controller
          );

  return Status;

}
```

Example 128. Supported() Service for USB Host Controller Driver

15.1.2.2 Start() and Stop()

The **Start()** service of the Driver Binding Protocol for the USB host controller driver also opens the PCI I/O Protocol **BY_DRIVER**. It will initialize the host controller and publish an instance of the **EFI_USB2_HC_PROTOCOL**.

Some USB host controllers provide legacy support to be compatible with legacy devices. Under this mode, the USB input device, including mouse and keyboard, will act as if they are behind an 8042 keyboard controller. In an EFI implementation, we use native USB support instead of this legacy support. As a result, in the **Start()** service of the USB host controller driver, the USB legacy support must be turned off before enabling the host controller. This step is required because the legacy support will conflict with the native USB support that's in the EFI USB driver stack.

Example 129 shows how to turn off USB legacy support for UHCI 1.1 host controllers.

```
//
// USB legacy Support
//
#define USB_EMULATION               0xc0

VOID
TurnOffUSBLegacySupport (
  IN EFI_PCI_IO_PROTOCOL  *PciIo
  )
{
  UINT16  Command;
  //
  // Disable USB Legacy Support
  //
  Command = 0;
  PciIo->Pci.Write (
          PciIo,
          EfiPciIoWidthUint16,
          USB_EMULATION,
          1,
          &Command
         );
  return;
}
```

Example 129. Turning off USB Legacy Support

The **Stop()** service will do the reverse of the steps that the **Start()** service does. The USB host controller driver needs to make sure that there are no memory leaks, as well as making sure that hardware is stopped accordingly.

15.1.3 USB2 Host Controller Protocol Transfer Related Services

The USB2 Host Controller Protocol provides an I/O abstraction for a USB host controller. A USB host controller is a hardware component that interfaces to a Universal Serial Bus (USB). It moves data between system memory and devices on the Universal Serial Bus by processing data structures and generating transactions on the Universal Serial Bus. This protocol is used by a USB bus driver to perform all data transaction over the Universal Serial Bus. It also provides services to manage the USB root hub that is integrated into the USB host controller.

Example 130 below shows a template for the implementation of the USB Host Controller Protocol. **<<DriverName>>** denotes the name of the USB host controller driver—for example, UHCI, EHCI, or OHCI. **<<UsbSpecificationMajorRevision>>** denotes the major revision of the *USB Specification* that the host controller follows. For example, for *USB 1.1 Specification*, it will be 1. **<<UsbSpecificationMinorRevision>>** denotes the minor revision of that *USB Specification.* For example, for the *USB 1.1 Specification*, it will be 1.

```
EFI_USB2_HC_PROTOCOL g<<DriverName>>UsbHc = {
  <<DriverName>>GetCapability,
  <<DriverName>>Reset,
  <<DriverName>>GetState,
  <<DriverName>>SetState,
  <<DriverName>>ControlTransfer,
  <<DriverName>>BulkTransfer,
  <<DriverName>>AsyncInterruptTransfer,
  <<DriverName>>SyncInterruptTransfer,
  <<DriverName>>IsochronousTransfer,
  <<DriverName>>AsyncIsochronousTransfer,
  <<DriverName>>GetRootHubPortStatus,
  <<DriverName>>SetRootHubPortFeature,
  <<DriverName>>ClearRootHubPortFeature,
  <<UsbSpecificationMajorRevision>>,
  <<UsbSpecificationMinorRevision>>,
};

EFI_STATUS
<<DriverName>>GetCapability (
  IN EFI_USB2_HC_PROTOCOL        *This,
  OUT UINT8                            *MaxSpeed,
  OUT UINT8                            *PortNumber,
  OUT UINT8                            *Is64BitCapable
)
{
}

EFI_STATUS
<<DriverName>>Reset (
  IN EFI_USB2_HC_PROTOCOL           *This,
  IN UINT16                         Attributes
  )
{
}

EFI_STATUS
<<DriverName>>GetState (
  IN  EFI_USB2_HC_PROTOCOL          *This,
  OUT EFI_USB2_HC_STATE              *State
  )
{
}

EFI_STATUS
<<DriverName>>SetState (
  IN EFI_USB2_HC_PROTOCOL           *This,
  IN EFI_USB2_HC_STATE               State
  )
{
}

EFI_STATUS
<<DriverName>>ControlTransfer (
  IN     EFI_USB2_HC_PROTOCOL       *This,
  IN     UINT8                      DeviceAddress,
  IN     UINT8                      DeviceSpeed,
  IN     UINT8                      MaximumPacketLength,
  IN     EFI_USB_DEVICE_REQUEST     *Request,
  IN     EFI_USB_DATA_DIRECTION     TransferDirection,
  IN OUT VOID                       *Data                  OPTIONAL,
  IN OUT UINTN                      *DataLength            OPTIONAL,
  IN     UINTN                      TimeOut,
  IN     EFI_USE2_HC_TRANSACTION_TRANSLATOR
*Translator,
  OUT    UINT32                     *TransferResult
  )
{
```

```
}

EFI_STATUS
<<DriverName>>BulkTransfer (
  IN  EFI_USB2_HC_PROTOCOL          *This,
  IN  UINT8                         DeviceAddress,
  IN  UINT8                         EndPointAddress,
  IN  UINT8                         DeviceSpeed,
  IN  UINT8                         MaximumPacketLength,
  IN  UINT8                         DataBuffersNumber,
  IN OUT VOID                       *Data[EFI_USB_MAX_BULK_BUFFER_NUM],
  IN OUT UINTN                      *DataLength,
  IN OUT UINT8                      *DataToggle,
  IN  UINTN                         TimeOut,
  IN  EFI_USE2_HC_TRANSACTION_TRANSLATOR
*Translator,
  OUT UINT32                        *TransferResult
  )
{
}

EFI_STATUS
<<DriverName>>AsyncInterruptTransfer (
  IN     EFI_USB2_HC_PROTOCOL             *This,
  IN     UINT8                            DeviceAddress,
  IN     UINT8                            EndPointAddress,
  IN     UINT8                            DeviceSpeed,
  IN     UINT8                            MaximumPacketLength,
  IN     BOOLEAN                          IsNewTransfer,
  IN OUT UINT8                            *DataToggle,
  IN     UINTN                            PollingInterval  OPTIONAL,
  IN     UINTN                            DataLength       OPTIONAL,
  IN     EFI_ASYNC_USB_TRANSFER_CALLBACK  CallBackFunction OPTIONAL,
  IN     VOID                             *Context         OPTIONAL
  )
{
}

EFI_STATUS
<<DriverName>>SyncInterruptTransfer (
  IN     EFI_USB2_HC_PROTOCOL       *This,
  IN     UINT8                      DeviceAddress,
  IN     UINT8                      EndPointAddress,
  IN     UINT8                      DeviceSpeed,
  IN     UINT8                      MaximumPacketLength,
  IN OUT VOID                       *Data,
  IN OUT UINTN                      *DataLength,
  IN OUT UINT8                      *DataToggle,
  IN     UINTN                      TimeOut,
  OUT    UINT32                     *TransferResult
  )
{
}

EFI_STATUS
<<DriverName>>IsochronousTransfer (
  IN     EFI_USB2_HC_PROTOCOL       *This,
  IN     UINT8                      DeviceAddress,
  IN     UINT8                      EndPointAddress,
  IN     UINT8                      DeviceSpeed,
  IN     UINT8                      MaximumPacketLength,
  IN     UINT8                      DataBuffersNum,
  IN OUT VOID                       *Data[EFI_USB_MAX_ISO_BUFFER_NUM],
  IN OUT UINTN                      DataLength,
  IN     EFI_USB2_HC_TRANSACTION_TRANSLATOR
*Translator,
  OUT    UINT32                     *TransferResult
  );
```

```
{
}

EFI_STATUS
<<DriverName>>AsyncIsochronousTransfer (
  IN     EFI_USB2_HC_PROTOCOL          *This,
  IN     UINT8                         DeviceAddress,
  IN     UINT8                         EndPointAddress,
  IN     UINT8                         DeviceSpeed,
  IN     UINT8                         MaximumPacketLength,
  IN     UINT8                         DataBuffersNum,
  IN OUT VOID                         *Data[EFI_USB_MAX_ISO_BUFFER_NUM],
  IN     UINTN                         DataLength,
  IN     EFI_USB2_HC_TRANSACTION_TRANSLATOR
*Translator,
  IN EFI_ASYNC_USB_TRANSFER_CALLBACK  IsochronousCallBack,
  IN VOID                             *Context   OPTIONAL
  )
{
}

EFI_STATUS
<<DriverName>>GetRootHubPortStatus (
  IN  EFI_USB2_HC_PROTOCOL           *This,
  IN  UINT8                          PortNumber,
  OUT EFI_USB_PORT_STATUS            *PortStatus
  )
{
}

EFI_STATUS
<<DriverName>>SetRootHubPortFeature (
  IN EFI_USB2_HC_PROTOCOL            *This,
  IN UINT8                           PortNumber,
  IN EFI_USB_PORT_FEATURE            PortFeature
  )
{
}

EFI_STATUS
<<DriverName>>ClearRootHubPortFeature (
  IN EFI_USB2_HC_PROTOCOL            *This,
  IN UINT8                           PortNumber,
  IN EFI_USB_PORT_FEATURE            PortFeature
  )
{
}
```

Example 130. Implementing the USB2 Host Controller Protocol

The interfaces of USB2 Host Controller Protocol can be categorized into the following aspects:

- Host controller general information
- **`GetCapability()`**
- Root hub–related services:
- **`GetRootHubPortStatus()`**
- **`SetRootHubPortFeature()`**
- **`ClearRootHubPortFeature()`**
- Host controller state–related services:

- **GetState()**
- **SetState()**
- **Reset()**
- USB transfer–related services:
- **ControlTransfer()**
- **BulkTransfer()**
- **AsyncInterruptTransfer()**
- **SyncInterruptTransfer()**
- **IsochronousTransfer()**
- **AsyncIsochronousTransfer()**

For implementing root hub–related services and host controller state–related services, it mainly involves read/write operations to specific USB host controller registers. The USB host controller data sheet provides abundant information on these register usages, so this topic will not be covered in detail here.

This section concentrates on the USB transfer–related services. We categorize those transfers as either *asynchronous* or *synchronous.*

The asynchronous transfer means the transfer will not complete with the service's return. The synchronous transfer means that, when the service returns, the transfer will also be complete. The following sections discuss these two types of transfers in more detail.

15.1.3.1 Synchronous Transfer

The USB Host Controller Protocol provides the following four synchronous transfer services:

- **ControlTransfer()**
- **BulkTransfer()**
- **SyncInterruptTransfer()**
- **IsochronousTransfer()**

Control and bulk transfers will be done in an acceptable period of time and thus are natural synchronous transfers in the view of an EFI system. Interrupt transfers and isochronous transfers can be either asynchronous or synchronous transfers, depending on the usage model.

It is convenient for the USB drivers to use these synchronous transfer services because they do not have to worry about when the data will be ready. The transfer result will be available as soon as the function returns.

The following is an example of how to use **BulkTransfer()** to implement a synchronous transfer service. Generally speaking, to implement a bulk transfer service, it can be divided into the following steps:

Preparation:

For example, USBSTS is a status register in the USB host controller. The status register needs to be cleared before starting the control transfer.

Setting up the DMA direction:

By judging the end point address, the USB driver will decide the transfer direction and then set up the PCI bus master read or write. For example, if the transfer direction is **`EfiUsbDataIn`**, then the USB host controller will read from the DMA buffer. So the bus master write needs to be set.

Building the transfer context:

The *USB Specification* defines several structures for a transfer. For example, Queue Head (QH) and Transfer Descriptor (TD) are special structures that are used to support the requirements of control, bulk, and interrupt transfers.

In this step, these QH and TD structures need to be created and linked to the Frame List. One possible implementation can be to create one QH and a list of TDs to form a transfer list. The QH will point to the first TD and occupy one entry in the Frame List.

Executing the TD and getting the result:

The USB host controller will automatically execute the TD when the timer comes. The UHCI driver needs to wait until the TDs of this transfer are all executed. After that, it will get the result of the TD execution.

Cleaning up:

It will delete the bulk transfer QH and TD structures from the Frame List, free related structures, and unregister the PCI DMA map.

15.1.3.2 Asynchronous Transfer

The USB Host Controller Protocol provides the following two asynchronous transfer services:

- **`AsyncInterruptTransfer()`**
- **`AsyncIsochronousTransfer()`**

To support asynchronous transfers, the USB host controller driver will register a periodical timer event. Meanwhile, it will maintain a queue for all asynchronous transfers. When the timer event is signaled, the timer event callback function will go through this queue and check whether some asynchronous transfers are complete.

Generally speaking, the main work of that timer event callback function is to go through the asynchronous transfers queue. For each asynchronous transfer, it will check whether that asynchronous transfer is completed or not and do the following:

- **If it is not completed**

  The USB host controller driver will take no action and still keep this transfer on the queue.

If it is completed

The USB host controller driver will copy the data that it received to a predefined data buffer and remove the related QH and TD structures. Meanwhile, it will also invoke a preregistered transfer callback function. Moreover, based on that transfer's complete status, the USB host controller driver will take different additional actions, as follows:

- If it completed without an error, it will update the transfer data status accordingly, e.g., data toggle bit.
- If it completed with an error, it is suggested that the USB host controller do nothing and leave the error recovery work to the related USB device driver.

15.1.3.3 Internal Memory Management

To implement USB transfers, the USB host controller driver needs to manage many small memory fragments as transfer data, for example, QH and TD. If the USB host controller driver uses the system memory management services to allocate these memory fragments each time, it is not efficient and fast enough. Thus, it is recommended that the USB host controller driver manage these kinds of internal memory usage itself. One possible implementation, as in the *EDK*, is that the host controller driver can allocate a large chunk of memory in its entry point by using EFI memory services. Then it will implement a small memory management algorithm to manage this memory to satisfy internal memory allocations. By using this simple memory management mechanism, it avoids the frequent system memory management calls.

15.1.3.4 DMA

Most USB host controllers use DMA for their data transfer between host and devices. Because the processor and USB host controller both need to access that transfer data simultaneously, the USB host controller driver shall use a common buffer for all the memory that the host controller uses for data transfer. This requirement means that the processor and the host controller have an identical view of memory. See chapter 14 for usage guidelines for the common buffer.

15.2 USB Bus Driver

The *EDK* contains a generic USB bus driver. This driver uses the services of **EFI_USB2_HC_PROTOCOL** to enumerate USB devices and produce child handles with **EFI_DEVICE_PATH_PROTOCOL** and **EFI_USB_IO_PROTOCOL**.

A USB hub, including the USB root hub and common hub, is a type of USB device. The USB bus driver is responsible for the management of all USB hub devices. No device drivers are required for USB hub devices.

If EFI-based system firmware is ported to a new platform, most of the USB-related changes occur in the implementation of the USB host controller driver. Moreover, to support additional USB devices, new USB device drivers are also required. However, the USB bus driver is designed to be a generic, platform-agnostic driver. As a result,

customizing the USB bus driver is strongly discouraged. The design and implementation of the USB bus driver will not be covered in detail in this document.

15.3 USB Device Driver

USB device drivers will use services provided by **EFI_USB_IO_PROTOCOL** to produce one or more protocols that provide I/O abstractions of a USB device. USB device drivers should follow the UEFI Driver Model. As mentioned above, the USB device drivers will not manage hub devices because those hub devices will be managed by a USB bus driver.

15.3.1 Sample USB device drivers in the EDK

- USB keyboard driver is located at **\Sample\Bus\Usb\UsbKb\Dxe**
- USB mouse driver is located at **\Sample\Bus\Usb\UsbMouse\Dxe**
- USB mass storage driver is located at **\Sample\Bus\Usb\UsbMassStorage\Dxe**

15.3.2 Driver Binding Protocol

USB device drivers are completely separate from USB host controller drivers and must separately meet all the UEFI Driver Morel Driver requirements. The main requirement in this context is supporting Driver Bindong Protocol.

15.3.2.1 Supported()

A USB device driver must implement the **EFI_DRIVER_BINDING_PROTOCOL** that contains the **Supported()**, **Start()**, and **Stop()** services. The **Supported()** service will check the controller handle that has been passed in to see whether this handle represents a USB device that this driver knows how to manage.

The following is the most common method for doing the check:

- Check if this handle has **EFI_USB_IO_PROTOCOL** installed. If not, this handle is not a USB device on the current USB bus.
- Get the USB interface descriptor back from this **USB_IO_DEVICE**. Check whether the values of this device's *InterfaceClass*, *InterfaceSubClass*, and *InterfaceProtocol* are identical to the corresponding values that this driver could manage.

If the above two checks are passed, it means that the USB device driver can manage the device that the controller handle represents. The **Supported()** service will return **EFI_SUCCESS**. Otherwise, the **Supported()** service will return **EFI_UNSUPPORTED**. In addition, this check process must not disturb the current state of the USB device because a different USB device driver may be controlling this USB device.

Example 131 is a fragment of code that shows how to implement a **Supported()** service for the USB device driver that manages a USB XYZ device with specific values for those critical fields.

```
EFI_STATUS
USBXYZDriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL    *This,
  IN EFI_HANDLE                     Controller,
  IN EFI_DEVICE_PATH_PROTOCOL       *RemainingDevicePath
  )
{
  EFI_STATUS            OpenStatus;
  EFI_USB_IO_PROTOCOL   *UsbIo;
  EFI_STATUS            Status;
  EFI_USB_INTERFACE_DESCRIPTOR    InterfaceDescriptor;

  //
  // Check if USB_IO protocol is attached on the controller handle.
  //
  OpenStatus = gBS->OpenProtocol (
                      Controller,
                      &gEfiUsbIoProtocolGuid,
                      &UsbIo,
                      This->DriverBindingHandle,
                      Controller,
                      EFI_OPEN_PROTOCOL_BY_DRIVER
                  );
  if (EFI_ERROR (OpenStatus)) {
    return OpenStatus;
  }

  //
  // Get the default interface descriptor
  //
  Status = UsbIo->UsbGetInterfaceDescriptor(
                      UsbIo,
                      &InterfaceDescriptor
                      );
  if(EFI_ERROR(Status)) {
    Status =  EFI_UNSUPPORTED;
  }    else {
    //
    // Judge whether the interface descriptor is supported by this driver
    //
    if (InterfaceDescriptor.InterfaceClass == CLASS_XYZCLASS &&
      InterfaceDescriptor.InterfaceSubClass == SUBCLASS_XYZSUBCLASS &&
      InterfaceDescriptor.InterfaceProtocol == PROTOCOL_XYZPROTOCOL) {
        Status =  EFI_SUCCESS;
    } else {
     Status =  EFI_UNSUPPORTED;
   }
 }


   gBS->CloseProtocol (
         Controller,
         &gEfiUsbIoProtocolGuid,
         This->DriverBindingHandle,
         Controller
         );

  return Status;
}
```

Example 131. Supported() Service of USB Device Driver

Because the **Supported()** service will be invoked many times, the USB bus driver in the *EDK* makes certain optimizations. It caches the interface descriptors, so that they do not have to read from the USB devices every time a USB device driver's **Supported()** service is invoked.

15.3.2.2 Start() and Stop()

The **Start()** service of the Driver Binding Protocol for a USB device driver will open the USB I/O Protocol **BY_DRIVER** and install the I/O abstraction protocol for the USB device onto the handle on which the **EFI_USB_IO_PROTOCOL** is installed.

This section provides detailed guidance on how to implement a USB device driver. It uses a USB mass storage mass storage device as an example. Suppose this mass storage device has the following four endpoints:

- One control endpoint
- One interrupt endpoint
- Two bulk endpoints

For the interrupt endpoint, it is synchronous. For the bulk endpoints, one is an input endpoint and the other is an output endpoint. The following sections will cover how to implement the **Start()** and **Stop()** driver binding protocol services and EFI Block I/O protocol.

Example 132 lists the private context data structure for this USB mass storage device driver. See chapter 8 for more details on private context data structure design guidelines.

```
typedef struct {
  UINT32            Signature;

  EFI_BLOCK_IO_PROTOCOL          BlockIO;
  EFI_USB_IO_PROTOCOL            *UsbIo;

  EFI_USB_INTERFACE_DESCRIPTOR   InterfaceDescriptor;
  EFI_USB_ENDPOINT_DESCRIPTOR    BulkInEndpointDescriptor;
  EFI_USB_ENDPOINT_DESCRIPTOR    BulkOutEndpointDescriptor;
  EFI_USB_ENDPOINT_DESCRIPTOR    InterruptEndpointDescriptor;

  ... ...
} USB_MASS_STORAGE_DEVICE;
```

Example 132. Implementing a USB CBI Mass Storage Device Driver

15.3.2.2.1 Implementing the DriverBinding.Start() Service

The code skeleton for the **Start()** service will be implemented in the following steps:

1. Open the USB I/O Protocol on *ControllerHandle* **BY_DRIVER**.
2. Get the interface descriptor using the **EFI_USB_IO_PROTOCOL.UsbGetInterfaceDescriptor()** service.
3. Prepare the private data structure.

This private data structure is in type **USB_MASS_STORAGE_DEVICE** and has fields for the interface descriptor, endpoint descriptor, and others.

This step will allocate memory for the private data structure and do the necessary initializations—for example, setting up the *Signature*, *UsbIo*, and *InterfaceDescriptor* fields.

4. Parse the interface descriptor.

   In this step, it will parse the *InterfaceDescriptor* that was obtained in step 2 and verify that all bulk and interrupt endpoints exit. The *NumEndpoints* field in *InterfaceDescriptor* indicates how many endpoints are in this USB interface. This code piece will first get endpoint descriptors one by one by using the **UsbGetEndpointDescriptor()** service. Then it will use the *Attributes* and *EndpointAddress* fields in *EndpointDescriptor* to judge the type of the endpoint.

5. Install the Block I/O protocol.

15.3.2.2.2 DriverBinding.Stop()

The **Stop()** service will do the reverse steps as the **Start()** service. It will uninstall the Block I/O Protocol and close its control to the USB I/O Protocol. It will also free various allocated resources—for example, the private data structure.

15.3.3 Asynchronous Transfer Usage

Example 133 shows how the USB device driver uses asynchronous transfers. It uses a USB mouse driver as an example and will use the asynchronous interrupt transfer to get mouse input.

In the USB mouse driver's Driver Binding Protocol **Start()** service, it will first initiate an asynchronous interrupt transfer.

```
Status = UsbIo->UsbAsyncInterruptTransfer(
                      UsbIo,
                      EndpointAddr,
                      TRUE,
                      PollingInterval,
                      PacketSize,
                      OnMouseInterruptComplete,
                      UsbMouseDevice
                    );
```

Example 133. Initiating an Asynchronous Interrupt Transfer in a USB Mouse Driver

In Example 134, **OnMouseInterruptComplete()** is the corresponding asynchronous interrupt transfer callback function. In this function, if the passing *Result* parameter indicates an error, it will clear the endpoint error status, unregister the previous asynchronous interrupt transfer, and initiate another asynchronous interrupt transfer. If there is no error, it will set the mouse state change indicator to **TRUE** and put the data that is read into the appropriate data structure.

```
EFI_STATUS
OnMouseInterruptComplete (
  IN  VOID        *Data,
  IN  UINTN       DataLength,
  IN  VOID        *Context,
  IN  UINT32      Result
  )
{
  USB_MOUSE_DEV       *UsbMouseDev;
  EFI_USB_IO_PROTOCOL *UsbIo;
  UINT8               EndpointAddr;
  UINT32              UsbResult;

  UsbMouseDev = (USB_MOUSE_DEV *)Context;
  UsbIo = UsbMouseDev->UsbIo;

  if (Result != EFI_USB_NOERROR) {
    if ((Result & EFI_USB_ERR_STALL) == EFI_USB_ERR_STALL) {
      EndpointAddr = UsbMouseDev->IntEndpointDescriptor->EndpointAddress;

      UsbClearEndpointHalt(
        UsbIo,
        EndpointAddr,
        &UsbResult
      );
    }

//
// Unregister previous asynchronous interrupt transfer
//
    UsbIo->UsbAsyncInterruptTransfer(
             UsbIo,
             UsbMouseDev->IntEndpointDescriptor->EndpointAddress,
             FALSE,
             0,
             0,
             NULL,
             NULL
           );

//
// Initiate a new asynchronous interrupt transfer
//
    UsbIo->UsbAsyncInterruptTransfer(
                     UsbIo,
                     UsbMouseDev->IntEndpointDescriptor->EndpointAddress,
                     TRUE,
                     UsbMouseDev->IntEndpointDescriptor->Interval,
                     UsbMouseDev->IntEndpointDescriptor->MaxPacketSize,
                     OnMouseInterruptComplete,
                     UsbMouseDev
                 );
    return EFI_DEVICE_ERROR;
  }


  UsbMouseDev->StateChanged = TRUE;

  //
  // Parse HID data package
  // and extract mouse movements and coordinates to UsbMouseDev
  //
  ... ...


  return EFI_SUCCESS;
}
```

Example 134. Completing an Asynchronous Interrupt Transfer

Example 135 shows the **GetMouseState()** service of the Simple Pointer In Protocol that the USB mouse driver will publish. **GetMouseState()**will not initiate any asynchronous interrupt transfer. It simply checks the mouse state change indicator. If there is mouse input, it will copy the mouse input to the passing *MouseState* data structure.

```
EFI_STATUS
GetMouseState(
  IN   EFI_SIMPLE_POINTER_PROTOCOL  *This,
  OUT  EFI_SIMPLE_POINTER_STATE     *MouseState
)
{
  USB_MOUSE_DEV           *MouseDev;


  MouseDev = USB_MOUSE_DEV_FROM_MOUSE_PROTOCOL(This);

  if (MouseDev->StateChanged == FALSE) {
    return EFI_NOT_READY;
  }

  EfiCopyMem(
    MouseState,
    &MouseDev->State,
    sizeof(EFI_SIMPLE_POINTER_STATE)
  );

  //
  // Clear previous move state
  //
  ... ...

  return EFI_SUCCESS;
}
```

Example 135. Retrieving Pointer Movement

15.3.4 State Machine Consideration

To implement USB device support, the USB device drivers need to maintain a state machine for their own transaction process. For example, the CBI driver for a USB mass storage device needs to maintain a tri-state machine, which contains Command->[Data]->Status states.

It should work well because it looks like a handshake process that is designed to be error free. Maintaining this state machine should provide enough and robust error handling.

However, imagine the following condition:

- A command is sent to the device that the host needs some data from the device.
- The device's response is too slow and it keeps NAK in its data endpoint.
- The host sees the NAK so many times that it thinks there will be no data available from the device. It will time out this data phase operation.

- The state machine is then in the status phase. It will ask for the status data from the device.
- The device then sends the real data phase data to the host.
- The host cannot understand the data from the device as status data, so it will reset the device and retry the operation.
- The necessary components of a dead loop then exist. The final result is a system likely to hang, an unusable device, or both.

How can this condition be avoided? If the device keeps NAK, it means that, sooner or later, the data will be available and no assumption can be made about the data's availability. There are some cases in which the device's response is so slow that the timeout is not enough for it to get data ready. As a result, retrying the transaction in the data phase may be necessary.

15.4 Debug Techniques

Several techniques can be used to debug the USB driver stack. The following sections describe these techniques.

15.4.1 Debug Message Output

One typical debug technique is to output the debug message. You can use the **DEBUG** macro to output debug message; see chapter 24 for the usage of the **DEBUG** macro. You can print the message both in the entry point and exit point of functions. By doing so, you can get the call stack and easily locate the error function. It is not suggested to print the debug message in a frequently called function, such as a timer handler.

15.4.2 USB Bus Analyzer

There are still some conditions where using the **DEBUG** macro is not enough for a developer to find the problem. One technique is to use a USB bus analyzer. Because a bus analyzer is inserted between the host and the device, the bus analyzer can get all the traffic on a single USB cable. With the USB bus's traffic information, some hard bugs can be root caused—for example, when a host controller loses packets on some occasions. Also, for the state machine chaos problem that was introduced in section 15.3.4, a bus analyzer will help to look at the packets' sequences and the unfinished state machine. The problem can also be quickly solved.

15.4.3 USBCheck/USBCV tool

Another useful tool for debugging is the USBCheck/USBCV tool from www.usb.org. This tool is very helpful when you want to see whether a device complies with the driver you are writing. Consider, for example, a case where a developer has written a USB imaging device driver for a generic imaging device such as a digital camera. If an end-user claims that this driver does not work for his or her specific brand of digital camera and the developer does not have such a camera on hand, the developer can ask the user to use the USBCheck/USBCV tool set and find out the device's *InterfaceClass*,

InterfaceSubClass, and *InterfaceProtocol*. The developer can then use this information to evaluate whether the camera should be supported by the driver.

15.5 Nonconforming Device

There are always arguments on how to deal with devices that do not conform to the *USB Specification.* It is suggested to stick to the specification and reject any nonconforming devices.

However, even if the device is nonconforming and the USB driver stack should reject it, developers need to make sure that the nonconforming device will not cause any system failures. The developer cannot make any assumptions about the device's behavior. It is essential for the end-user's experience that the nonconforming device does not negatively affect the system.

16 SCSI Driver Design Guidelines

This chapter focuses on the design and implementation of EFI SCSI drivers. Most SCSI controllers are PCI controllers, and the SCSI drivers managing them are also PCI drivers. As such, they must follow all of the design guidelines described in chapter 14, as well as the general guidelines described in chapter 5. In addition, this chapter covers the guidelines that apply specifically to the management of SCSI host controllers, SCSI channels, and SCSI devices.

16.1 SCSI Driver Overview

- Per the *UEFI 2.0 Specification*, the EFI SCSI driver is mainly referred as a SCSI host controller driver that manages a SCSI host controller that contains one or more SCSI channels. It may create SCSI channel handles for each SCSI channel and attach Ext SCSI Pass Thru Protocol and Device Path Protocol to each handle that the driver produced.

These I/O abstractions allow the SCSI device to be used in the preboot environment. See the *UEFI 2.0 Specification* for details about **EFI_EXT_SCSI_PASS_THRU_PROTOCOL**.

Note: *In order for the driver to function on an EFI 1.10 compliant system the SCSI driver is also responsible for producing Block I/O Protocol or other equivalent I/O abstraction protocols on the SCSI device handle. In this case the driver must build the entire driver stack including any required intermediate protocols.*

Note: *If the driver needs to perform correctly on both UEFI 2.0 and EFI 1.10 systems the system table should be examined to determine which method needs to be followed.*

16.2 SCSI Adapter Considerations

An EFI SCSI driver follows the UEFI Driver Model. Depending on the adapter that it manages, a SCSI driver can be categorized as either a device driver or a hybrid driver. It will create child handles for each SCSI channel (if there is more than 1) and it may also install protocols on its own handle. Typically, SCSI drivers are chip-specific because of the requirement to initialize and manage the currently bound SCSI device.

Because there may be multiple SCSI host adapters that can be managed by a single SCSI driver in the same platform, it is recommended that the SCSI host controller driver be designed to be re-entrant, as described in section 5.6 of this document.

16.2.1 Single-Channel SCSI Adapters

If the target SCSI adapter supports only one channel, then the SCSI driver can simply do the following:

- Attach **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** to itself (with both logical and physical bits on).

The following figure shows an example implementation on a single-channel SCSI adapter. Only the Green layer is from the SCSI Driver.

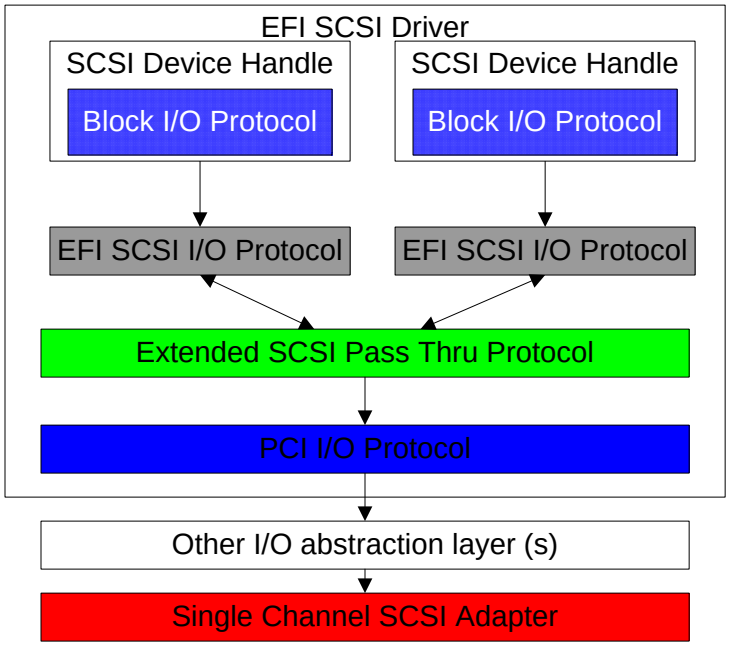

Figure 18. Sample SCSI Driver Stack on Single-Channel Adapter

Because there is only one SCSI channel, the SCSI driver can simply implement one instance of the Ext SCSI Pass Thru Protocol. The system firmware will then complete the driver stack by performing the following actions:

- Scan for SCSI device handles
- Attach SCSI IO to each handle
- Attach an appropriate IO abstraction (Like Block I/O)

This case is quite simple because the one SCSI pass thru–per–SCSI channel mapping is very clear.

16.2.2 Multichannel SCSI Adapters

An EFI SCSI driver becomes more complex if the SCSI adapter to be managed produces multiple SCSI channels. Figure 19 shows a possible SCSI driver implementation on a two-channel SCSI adapter.

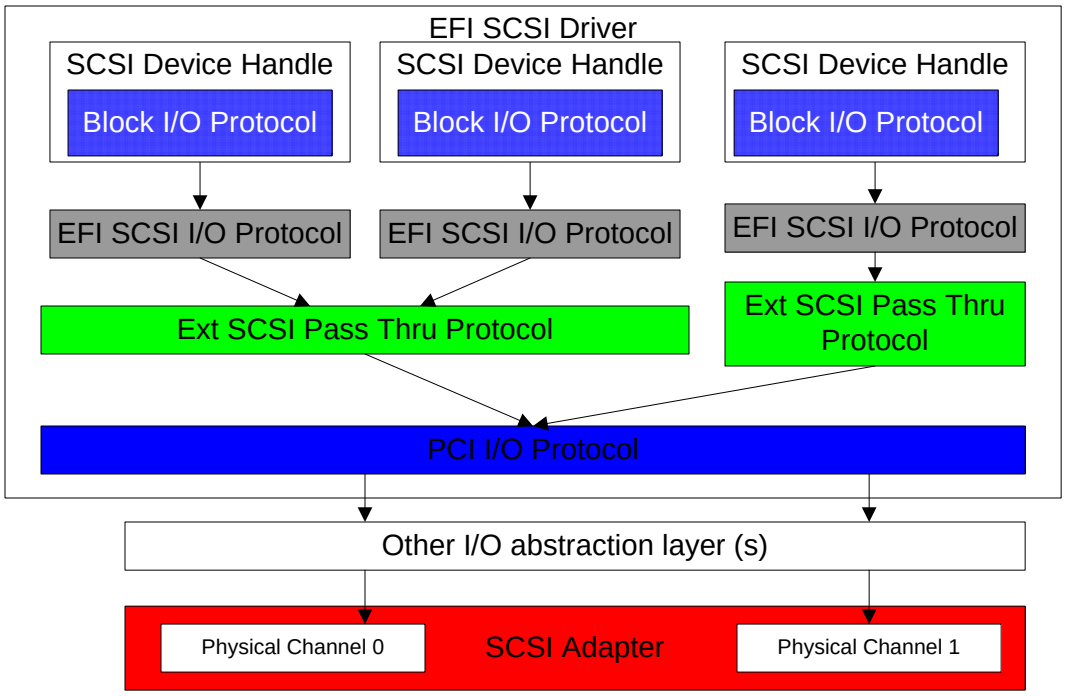

Figure 19. Sample SCSI Driver Implementation on a Multichannel Adapter

In this case, one SCSI adapter produces two physical SCSI channels. The SCSI driver should do the following:

- Create SCSI channel handles for each physical channel.
- Produce an instance of Ext SCSI Pass Thru Protocol (with physical and logical bits) for each of them.
- Attach all the instances of Ext SCSI Pass Thru Protocol to the handles.

The system firmware then completes the stack.

16.2.3 SCSI RAID Adapters

An EFI SCSI driver can also support SCSI adapters with RAID capability. Figure 20 shows an example implementation with two physical channels and one logical channel.

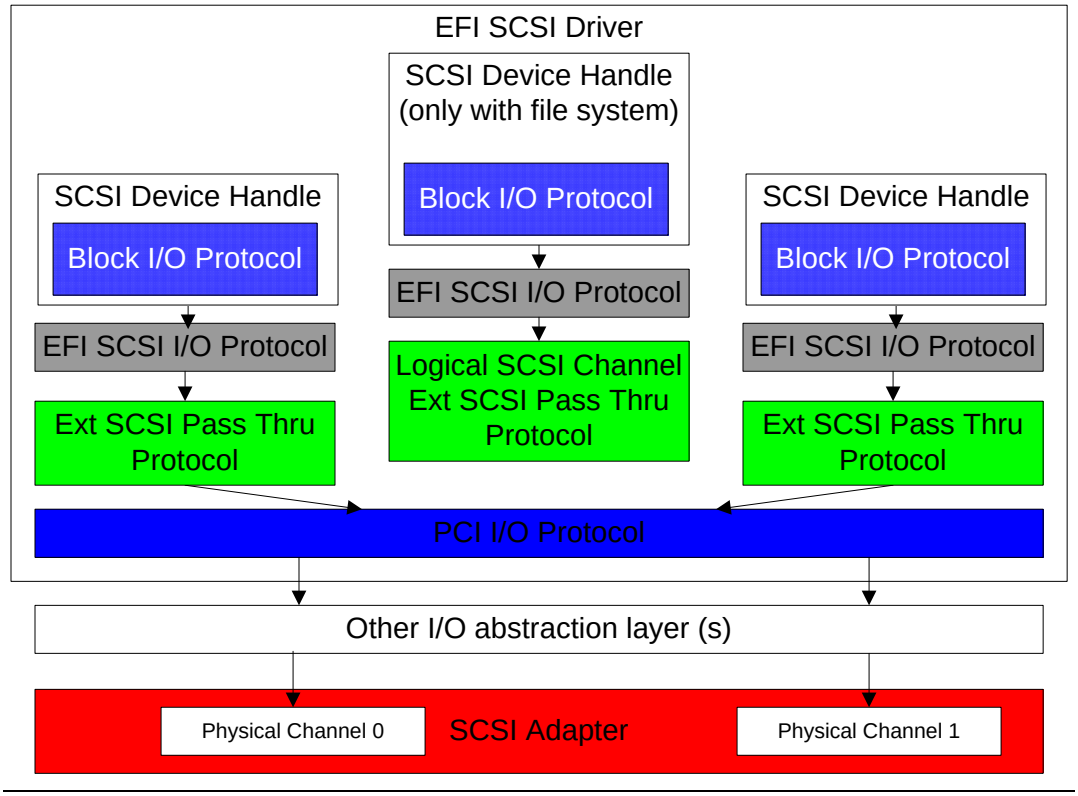

Figure 20. Sample SCSI Driver Implementation on Multichannel RAID Adapter

In this example, two physical channels are implemented on the SCSI adapter. The RAID component then configures these two channels to produce a logical SCSI channel. The two physical channels each have Ext SCSI Pass Thru installed, but these will not be used except for diagnostic use. For the logical channel, the SCSI driver will produce another Ext SCSI Pass Thru Protocol (with physical bit off) instance based on the RAID configuration. Messages passed to this protocol will be passed however the driver wants. SCSI devices on this logical channel should be the only ones enumerated by the system.

The SCSI adapter hardware may not be able to expose the physical SCSI channel(s) to upper-level software when implementing RAID. If the physical SCSI channel cannot be exposed to upper software, then the SCSI driver is required only to produce a single RAID logical channel.

Although the basic theory is the same as the one on a physical channel, it is different from a manufacturing and diagnostic perspective. If the physical SCSI channels are exposed, any SCSI command, including diagnostic ones, can be sent to an individual channel, which is very helpful on manufacturing lines. Furthermore, the diagnostic command can be sent simultaneously to all physical channels using the non-blocking mode that is supported by Ext SCSI Pass Thru Protocol. The diagnostic process may considerably benefit from the performance gain. In summary, it is suggested to expose physical SCSI channel whenever possible.

Of course, there are many possible designs for implementing SCSI RAID functionality. The point is that an EFI SCSI driver can be designed and implemented for a wide variety of SCSI adapters, and those EFI SCSI drivers can produce Ext SCSI Pass Thru Protocol for SCSI channels.

16.2.4 Implementing Driver Binding Protocol

Like many other drivers that follow the UEFI Driver Model, the image entry point of a SCSI driver installs the Driver Binding Protocol instance on the image handle. All three of the services in the Driver Binding Protocol—**Supported()**, **Start()**, and **Stop()**—must be implemented by an EFI SCSI driver.

16.2.4.1 Supported()

The **Supported()** function tests to see whether the given handle is a manageable SCSI adapter. In this function, a SCSI driver should check that **EFI_DEVICE_PATH_PROTOCOL** and **EFI_PCI_IO_PROTOCOL** are present to ensure the handle that is passed in represents a PCI device. In addition, a SCSI driver should also check the *ClassCode*, *VendorId*, and *DeviceId* that are read from the device's PCI configuration header to make sure it is a compliant SCSI adapter that can be managed by the EFI SCSI driver.

16.2.4.2 Start()

The **Start()** function tells the SCSI driver to start managing the SCSI controller. In this function, a single channel SCSI driver should use chip-specific knowledge to do the following:

- Initialize the SCSI host controller.
- Enable the PCI device.
- Allocate resources.
- Construct data structures for the driver to use.
- Implement the interfaces that are defined in **EFI_EXT_SCSI_PASS_THRU_PROTOCOL**.

If the SCSI adapter is a single-channel adapter, then the EFI SCSI driver should install **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** on the same handle that has the PCI I/O Protocol attached. If the SCSI adapter is a multi-channel adapter, then the driver should also do the following:

- Enumerate the SCSI channels that are supported by the host controller.
- Create SCSI channel handles for each detected SCSI channel.
- Append the device path for each channel handle.
- Attach **EFI_DEVICE_PATH_PROTOCOL** and **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** to every newly created channel handle.

16.2.4.3 Stop()

The **Stop()** function performs the opposite operations as **Start()**. Generally speaking, a SCSI driver is required to do the following:

- Disable the SCSI adapter.
- Release all resources that were allocated for this driver.
- Close the protocol instances that were opened in the **Start()**function.
- Uninstall the protocol interfaces that were attached on the host controller handle.

In general, if it is possible to design an EFI SCSI driver to create one child at a time, it should do so to support the rapid boot capability in the UEFI Driver Model. Each of the channel child handles created in **Start()** must contain a Device Path Protocol instance and a Ext SCSI Passthru abstraction layer. The format of device paths for SCSI devices is described in section 16.5.

16.2.5 Implementing Ext SCSI PassThru Protocol

EFI_EXT_SCSI_PASS_THRU_PROTOCOL allows information about a SCSI channel to be collected and allows SCSI Request Packets to be sent to any SCSI devices on a SCSI channel, even if those devices are not boot devices. This protocol is attached to the device handle of each SCSI channel in a system that the protocol supports and can be used for diagnostics. It may also be used to build a block I/O driver for SCSI hard drives and SCSI CD-ROM or DVD drives to allow those devices to become boot devices.

The protocol interface for Ext SCSI Pass Thru Protocol is listed below:

Protocol Interface Structure

```
typedef struct _EFI_EXT_SCSI_PASS_THRU_PROTOCOL {
EFI_EXT_SCSI_PASS_THRU_MODE                    *Mode;
EFI_EXT_SCSI_PASS_THRU_PASSTHRU                PassThru;
EFI_EXT_SCSI_PASS_THRU_GET_NEXT_TARGET_LUN     GetNextTargetLun;
EFI_EXT_SCSI_PASS_THRU_BUILD_DEVICE_PATHBuildDevicePath;
EFI_EXT_SCSI_PASS_THRU_GET_TARGET_LUN          GetTargetLun;
EFI_EXT_SCSI_PASS_THRU_RESET_CHANNEL           ResetChannel;
EFI_EXT_SCSI_PASS_THRU_RESET_TARGET_LUN        ResetTargetLun;
EFI_EXT_SCSI_PASS_THRU_GET_NEXT_TARGET         GetNextTarget;}
EFI_EXT_SCSI_PASS_THRU_PROTOCOL;
```

For a detailed description of **EFI_EXT_SCSI_PASS_THRU_PROTOCOL**, see chapter 14.8 of the *UEFI 2.0 Specification.*

Before implementing Ext SCSI Pass Thru Protocol, the SCSI driver should configure the SCSI core to a defined state. In practice, the SCSI adapter usually maps a set of SCSI core registers in I/O or memory-mapped I/O space. Although the detailed layout or functions of these registers vary from one SCSI hardware to another, the SCSI driver should use specific knowledge to set up the proper SCSI working mode (SCSI-I, SCSI-II, Ultra SCSI, and so on) and configure the timing registers for the current mode. Other considerations include parity options, DMA engine and interrupt initialization, among others.

All the hardware-related settings should be completed before any Ext SCSI Pass Thru Protocol function is called. The initialization is better accomplished in the Driver Binding Protocol's **Start()** function of the SCSI controller driver, prior to hooking up Ext SCSI Pass Thru Protocol functions.

EFI_EXT_SCSI_PASS_THRU_PROTOCOL.*Mode* is a structure that describes the intrinsic attributes of Ext SCSI Pass Thru Protocol instance. Note that a non-RAID SCSI channel should set both the physical and logical attributes. A physical channel on the RAID adapter should set only the physical attribute, and the logical channel on the RAID adapter should set only the logical attribute. If the channel supports non-blocking I/O, the non-blocking attribute should also be set. Example 136 shows how to set those attributes on a non-RAID SCSI adapter that supports non-blocking I/O.

```
ExtScsiPassThruMode.AdapterId = 4; // Target Channel Id
ExtScsiPassThruMode.Attributes =
EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL |
EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL |
EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO;
ExtScsiPassThruMode.IoAlign     = 0; // Do not have any alignment
requirement
```

Example 136. SCSI Pass Thru Mode Structure on Single-Channel SCSI Adapter

Example 137 shows how to set the SCSI *Mode* structure on a multi-channel non-RAID adapter. The example fits for either channel in Figure 19.

```
ExtScsiPassThruMode.AdapterId = 2;
// The channel does not support nonblocking I/O
ExtScsiPassThruMode.Attributes =
EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL |
EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL;
ExtScsiPassThruMode.IoAlign     = 2; // Data must be alligned on
                                     // 4-byte boundary
```

Example 137. SCSI Pass Thru Mode Structure on Multichannel SCSI Adapter

For the RAID adapter shown in Figure 20, the corresponding *Mode* structures for both the physical and logical channel may be filled as shown in Example 138.

```
// ... .. ... Physical Channel ... .. ...
ExtScsiPassThruMode.AdapterId = 0;
ExtScsiPassThruMode.Attributes =
EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL
| EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO;
ExtScsiPassThruMode.IoAlign     = 0;

//  ... .. ... Logical Channel ... .. ...
ExtScsiPassThruMode.AdapterId = 2;
ExtScsiPassThruMode.Attributes =
EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL
| EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO;
ExtScsiPassThruMode.IoAlign     = 0;
```

Example 138. SCSI Pass Thru Mode Structures on RAID SCSI Adapter

The **EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetNextTarget()** and **EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetTargetLun()** functions provide a way to walk on different devices within a channel. The SCSI controller driver may implement it by internally maintaining an active device flag. Use this flag and channel-specific knowledge to figure out what device is next, as well as what device is first.

The **EFI_EXT_SCSI_PASS_THRU_PROTOCOL.BuildDevicePath()** function facilitates the construction of a SCSI device path. The device path for the SCSI device can be many kinds because of enormous device types that are supported by the SCSI pass thru mechanism. The detailed device category can be identified only by the SCSI pass thru implementation, which is why the function goes into Ext SCSI Pass Thru Protocol. See the last section in this chapter for device path examples for SCSI devices.

The **EFI_EXT_SCSI_PASS_THRU_PROTOCOL.PassThru()** function is the most important function when implementing Ext SCSI Pass Thru Protocol. In this function, the SCSI pass thru driver should do the following:

- Initialize the internal register for command/data transfer.
- Put valid SCSI packets into hardware-specific memory or register locations.
- Start the transfer.
- Optionally wait for completion of the execution.

The better error handling mechanism in this function helps to develop a more robust driver. Although most SCSI adapters support both synchronous and asynchronous data transfers, some may not support synchronous mode. In this case, the SCSI driver may implement the blocking SCSI I/O that is required by the *UEFI 2.0 Specification* using the polling mechanism. Polling can be based on a timer interrupt or simply by polling the internal register. Do not return until all I/O requests are completed or else an unhandled error is encountered.

Example 139 below shows a template for implementing Ext SCSI Pass Thru Protocol.

```
// In the SCSI Channel private context data definition
typedef struct {
    UINTN                              Signature;
    EFI_HANDLE                         Handle;
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL    ScsiPassThru;
    EFI_EXT_SCSI_PASS_THRU_MODE        ScsiPassThruMode;
    // .. .. ..
} <<DriverName>>_EXT_SCSI_PASS_THRU_CONTEXT_DATA;

// In the SCSI driver implementation file
<<DriverName>>_EXT_SCSI_PASS_THRU_CONTEXT_DATA   *ScsiContextData;

Status = gBS->AllocatePool (
                  EfiBootServicesData,
                  sizeof(<<DriverName>>_EXT_SCSI_PASS_THRU_CONTEXT_DATA),
                 (VOID **)&ScsiContextData
                 );
if (EFI_ERROR(Status)) {
   return Status;
}
EfiZeroMem (
ScsiContextData,
sizeof (<<DriverName>>_EXT_SCSI_PASS_THRU_CONTEXT_DATA)
);
ScsiContextData->Signature =
<<DriverName>>_EXT_SCSI_PASS_THRU_DEV_SIGNATURE;
//
// Initialize Ext SCSI Pass Thru Protocol interface
//
ScsiContextData->ExtScsiPassThru.Mode = &ScsiContextData-
>ExtScsiPassThruMode;
ScsiContextData->ExtScsiPassThru.PassThru =
<<DriverName>>ExtScsiPassThruPassThru;
ScsiContextData->ExtScsiPassThru.GetNextTargetLun =
 <<DriverName>>ExtScsiPassThruGetNextTargetLun;
ScsiContextData->ExtScsiPassThru.BuildDevicePath  =
<<DriverName>>ExtScsiPassThru BuildDevicePath;
ScsiContextData->ExtScsiPassThru.GetTargetLun     =
<<DriverName>>ExtScsiPassThruGetTargetLun;
ScsiContextData->ExtScsiPassThru.ResetChannel      =
<<DriverName>>ExtScsiPassThruResetChannel;
ScsiContextData->ExtScsiPassThru.ResetTargetLun       =
<<DriverName>>ExtScsiPassThruResetTargetLun;
ScsiContextData->ExtScsiPassThru.GetNextTarget =
 <<DriverName>>ExtScsiPassThruGetNextTarget;

// In the SCSI Pass Thru implementation file
EFI_STATUS
<<DriverName>>ExtScsiPassThruPassThru (
  IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL              *This,
  IN UINT8                                        *Target,
  IN UINT64                                       Lun,
  IN OUT EFI_EXT_SCSI_PASS_THRU_SCSI_REQUEST_PACKET   *Packet,
  IN EFI_EVENT                                    Event OPTIONAL
  )
{
}
EFI_STATUS
<<DriverName>>ExtScsiPassThruGetNextTargetLun (
  IN  EFI_EXT_SCSI_PASS_THRU_PROTOCOL             *This,
  IN OUT UINT8                                    **Target,
  IN OUT UINT64                                   *Lun
  )
{
}
EFI_STATUS
<<DriverName>>ExtScsiPassThruBuildDevicePath (
  IN     EFI_EXT_SCSI_PASS_THRU_PROTOCOL          *This,
```

```
  IN     UINT8                                    *Target,
  IN     UINT64                                   Lun,
  IN OUT EFI_DEVICE_PATH_PROTOCOL                 **DevicePath
  )
{
}
EFI_STATUS
<<DriverName>>ExtScsiPassThruGetTargetLun (
  IN  EFI_EXT_SCSI_PASS_THRU_PROTOCOL             *This,
  IN  EFI_DEVICE_PATH_PROTOCOL                    *DevicePath,
  OUT UINT8                                       **Target,
  OUT UINT64                                      *Lun
  )
{
}
EFI_STATUS
<<DriverName>>ExtScsiPassThruResetChannel (
  IN  EFI_EXT_SCSI_PASS_THRU_PROTOCOL             *This
  )
{
}
EFI_STATUS
<<DriverName>>ExtScsiPassThru ResetTarget (
  IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL              *This,
  IN UINT8                                        *Target,
  IN UINT64                                       Lun
  )
{
}
EFI_STATUS
<<DriverName>>ExtScsiPassThruGetNextTarget (
  IN  EFI_EXT_SCSI_PASS_THRU_PROTOCOL             *This,
  IN OUT UINT8                                    **Target,
  )
{
}
```

Example 139. Ext SCSI Pass Thru Protocol Template

16.3 SCSI Command Set Device Considerations

Ext SCSI Pass Thru Protocol defines a method to directly access SCSI devices. This protocol provides interfaces that allow a generic driver to produce the Block I/O Protocol for SCSI disk devices and allows an UEFI utility to issue commands to any SCSI device. The main reason to provide such an access is to enable S.M.A.R.T. functionality during POST (i.e., issuing Mode Sense, Mode Select, and Log Sense to SCSI devices). This enabling is accomplished using the generic interfaces that are defined in Ext SCSI Pass Thru Protocol. The implementation of this protocol will also enable additional functionality in the future without modifying the SCSI drivers that are built on top of the SCSI driver. Furthermore, Ext SCSI Pass Thru Protocol is not limited to SCSI adapters. It is applicable to all channel technologies that use SCSI commands such as ATAPI, iSCSI, and Fibre Channel. This section shows some examples that demonstrate how to implement Ext SCSI Pass Thru Protocol on SCSI command set–compatible technology.

16.3.1 ATAPI

This section shows how to implement Ext SCSI Pass Thru Protocol for ATAPI devices.

Decode the Target and Lun pair using the intrinsic property of the technology or device. In this example, ATAPI supports only four devices, so the Target and Lun pair can be decoded by determining the IDE channel (primary/secondary) and IDE device (master/slave).

If the corresponding technology or device supports the channel reset operation, use it to implement **EFI_EXT_SCSI_PASS_THRU_PROTOCOL.ResetChannel()**; if not, it may be implemented by resetting all attached devices on the channel and re-enumerating them.

In the **EFI_EXT_SCSI_PASS_THRU_PROTOCOL.BuildDevicePath()** function, all target devices should be built on a node based on the channel knowledge. Example 140 shows an ATAPI node being built.

```
typedef struct {
    UINTN                               Signature;
    EFI_HANDLE                          Handle;
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL     ScsiPassThru;
    EFI_EXT_SCSI_PASS_THRU_MODE         ScsiPassThruMode;
    EFI_PCI_IO_PROTOCOL                 *PciIo;
    IDE_BASE_REGISTERS                  *IoPort;
    CHAR16                              ControllerName[100];
    CHAR16                              ChannelName[100];
    UINT32                              LatestTargetId;
    UINT64                              LatestLun;
} ATAPI_SCSI_PASS_THRU_DEV;

EFI_STATUS
SampleAtapiScsiPassThruBuildDevicePath (
  IN     EFI_EXT_SCSI_PASS_THRU_PROTOCOL    *This,
  IN     UINT8                              *Target,
  IN     UINT64                             Lun,
  IN OUT EFI_DEVICE_PATH_PROTOCOL           **DevicePath
  )
{
  ATAPI_SCSI_PASS_THRU_DEV    *AtapiScsiPrivate;
  EFI_DEV_PATH                *Node;
  EFI_STATUS                  Status;

// Checking parameters..........

  // Retrieve Device Private Data Structure via _CR() macro
  AtapiScsiPrivate = ATAPI_EXT_SCSI_PASS_THRU_DEV_FROM_THIS (This);

  Status = gBS->AllocatePool (
                    EfiBootServicesData,
                    sizeof(EFI_DEV_PATH),
                    (VOID **)&Node);
  if (EFI_ERROR(Status)) {
    return EFI_OUT_OF_RESOURCES;
  }

  gBS->SetMem (Node, sizeof(EFI_DEV_PATH), 0);
  Node->DevPath.Type            = MESSAGING_DEVICE_PATH;
  Node->DevPath.SubType         = MSG_ATAPI_DP;
  (&Node->DevPath)->Length[0]   = (UINT8) (sizeof(ATAPI_DEVICE_PATH));
  (&Node->DevPath)->Length[1]   = (UINT8) ((sizeof(ATAPI_DEVICE_PATH)) >>
8);
  Node->Atapi.PrimarySecondary = (UINT8)(Target / 2);
  Node->Atapi.SlaveMaster      = (UINT8)(Target % 2);
  Node->Atapi.Lun              = (UINT16)Lun;

  *DevicePath = (EFI_DEVICE_PATH_PROTOCOL*)Node;
  return EFI_SUCCESS;
}
```

Example 140. Building Device Path for ATAPI Device

For the most important function, **EFI_EXT_SCSI_PASS_THRU_PROTOCOL.PassThru()**, it should be implemented by technology-dependent means. In this example, ATAPI supports a SCSI command using the IDE "Packet" command. Because the IDE command is delivered through a group of I/O registers, the main body of the implementation is filling the SCSI command structure to these I/O registers and then waiting for the command completion. A complete code example for the blocking I/O **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** function can be found in the **\Sample\Bus\Pci\AtapiExtPassThru\Dxe** directory of the *EDK.*

For the non-blocking I/O **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** function, the SCSI driver should simply submit the SCSI command and return. It may choose to poll an internal timer event to check whether the submitted command completes its execution. If so, it should signal the client event. The EFI firmware will then schedule to invoke the notification function of the client event. Example 141 shows a sample non-blocking Ext SCSI Pass Thru Protocol implementation.

```
EFI_STATUS
SampleNonBlockingScsiPassThruFunction (
  IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL                  *This,
  IN UINT8                                            *Target,
  IN UINT64                                           Lun,
  IN OUT EFI_EXT_SCSI_PASS_THRU_SCSI_REQUEST_PACKET   *Packet,
  IN EFI_EVENT                                        Event OPTIONAL
  )
{
  ATAPI_SCSI_PASS_THRU_DEV        *AtapiScsiPrivate;
  EFI_EVENT                       InternalEvent;
  EFI_STATUS                      Status;

  AtapiScsiPrivate = ATAPI_SCSI_PASS_THRU_DEV_FROM_THIS (This);

  //
  // Do parameter checking required by UEFI specification
  //
  //..................................

  //
  // Create internal timer event in order to poll the completion.
  // The event can also be created outside of this function to
  // avoid frequent event construction/destruction.
  //
  Status = gBS->CreateEvent (
                EFI_EVENT_TIMER | EFI_EVENT_NOTIFY_SIGNAL,
                EFI_TPL_CALLBACK,
                ScsiPassThruPollEventNotify,
                AtapiScsiPrivate,
                &InternalEvent
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Signal the polling event every 200 ms. Select the interval
  // according to the specific requirement and technology.
  //
  Status = gBS->SetTimer (InternalEvent, TimerPeriodic, 2000000);
  if (EFI_ERROR (Status)) {
    return Status;
  }

  //
  // Just submit SCSI I/O command through IDE I/O registers and
  // return
  //
  Status = SubmitBlockingIoCommand (AtapiScsiPrivate, Target, Packet);
  return Status;
}

VOID
ScsiPassThruPollEventNotify (
  IN  EFI_EVENT                Event,
  IN  VOID                     *Context
  )
{
  ATAPI_SCSI_PASS_THRU_DEV     *AtapiScsiPrivate;
  BOOLEAN                      CommandCompleted;

  ASSERT (Context);
  AtapiScsiPrivate = (ATAPI_SCSI_PASS_THRU_DEV *)Context;
  CommandCompleted = FALSE;

  //
  // Use specific knowledge to identify whether command execution
```

```
  // completes or not. If so, set CommandCompleted as TRUE.
  //
  // ......................

  if (CommandCompleted) {
    //
    // Get client event handle from private context data structure.
    // Signal it.
    //
    gBS->SignalEvent (ClientEvent);
  }
}
```

Example 141. Sample Non-blocking Ext SCSI Pass Thru Protocol Implementation

16.4 Discover a SCSI channel

It is recommended that the SCSI driver constructs a private context structure for each enumerated SCSI channel. See chapter 8 in this document for the advantage of using such a private context structure. Specifically, the SCSI driver should store all required information for the child SCSI channel in this data structure, this should including the signature, child handle value (optional for single channel controller), channel number, and any produced protocols. This private context structure can be accessed via the Record macro **_CR()**, which can also be found in chapter 8 of this document.

The method for determining the number of channels on a given controller is silicon specific and varies by manufacturer.

It is also the SCSI driver's responsibility to do the following:

- Build the appropriate device path for the enumerated SCSI channel.
- Install Ext SCSI Passthru protocol and Device Path Protocol on the appropriate handle (child handle is optional for single channel).

16.5 SCSI Device Path

The SCSI driver described in this document can support a SCSI channel that is generated or emulated by multiple architectures, such as SCSI-I, SCSI-II, SCSI-III, ATAPI, Fibre Channel, iSCSI, and other future channel types. This section describes four example device paths, including SCSI, ATAPI, and Fibre Channel device paths.

16.5.1 SCSI Device Path Example

Table 29 shows an example device path for a SCSI device controller on a desktop platform. This SCSI device controller is connected to a SCSI channel that is generated by a PCI SCSI host controller. The PCI SCSI host controller generates a single SCSI channel, is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge.

This sample device path consists of an ACPI device path node, a PCI device path node, and a device path end structure. The _HID and _UID must match the ACPI table

description of the PCI root bridge. The following is the shorthand notation for this device path: **ACPI(PNP0A03,0)/PCI(7|0)**

Table 29. SCSI Device Path Examples

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x02	Generic Device Path Header – Type ACPI Device Path
0x01	0x01	0x01	Sub type – ACPI Device Path
0x02	0x02	0x0C	Length – 0x0C bytes
0x04	0x04	0x41D0, 0x0A03	_HID PNP0A03 – 0x41D0 represents a compressed string 'PNP' and is in the low-order bytes.
0x08	0x04	0x0000	_UID
0x0C	0x01	0x01	Generic Device Path Header – Type Hardware Device Path
0x0D	0x01	0x01	Sub type – PCI
0x0E	0x02	0x06	Length – 0x06 bytes
0x10	0x01	0x07	PCI Function
0x11	0x01	0x00	PCI Device
0x12	0x01	0xFF	Generic Device Path Header – Type End of Hardware Device Path
0x13	0x01	0xFF	Sub type – End of Entire Device Path
0x14	0x02	0x04	Length – 0x04 bytes

16.5.2 Multiple SCSI channels on a Multifunction PCI Controller

A SCSI device on a multi-channel PCI controller only changes the PCI portion of the device path. In this example, physical SCSI device 0 is attached on physical channel 1 of a multifunction PCI SCSI controller. SCSI channel 0 is accessed through PCI function #0, and SCSI channel 1 is accessed through PCI function #1. The following are the device paths for these SCSI channels:

ACPI(PNP0A03,1)/PCI(7|0) Access to channel 0

ACPI(PNP0A03,1)/PCI(7|1) Access to channel 1

16.5.3 Multiple SCSI channels on a single function PCI controller

If there is a SCSI PCI controller with multiple SCSI channels connected to a one function PCI device the device paths are different. Physical SCSI device 0 is attached on physical channel 1 of the PCI SCSI controller. Both SCSI channels are on the same function, but are differentiated by the Controller node.

ACPI(PNP0A03,1)/PCI(7|0)/Controller(0) Access to channel 0

ACPI(PNP0A03,1)/PCI(7|0)/Controller(1) Access to channel 1

16.6 Using Ext SCSI Pass Thru Protocol

If a SCSI driver supports both blocking and non-blocking I/O modes, any client of the SCSI driver can use them to perform SCSI I/O. Example 142 demonstrates how to use Ext SCSI Pass Thru Protocol to perform blocking and non-blocking I/O.

```
EFI_STATUS
ScsiPassThruTests(
  EFI_EXT_SCSI_PASS_THRU_PROTOCOL  *EfiSptProtocol
  )
{
  EFI_STATUS                                Status;
  UINT8                                     *Target;
  UINT64                                    Lun;
  EFI_EXT_SCSI_PASS_THRU_SCSI_REQUEST_PACKET Packet;
  EFI_EVENT                                 Event;

  . . .

  //
  // Blocking I/O
  //
  Status = EfiSptProtocol->PassThru (
                                   EfiSptProtocol,
                                   Target,
                                   Lun,
                                   &Packet,
                                   NULL
                                   );
  . . .

  //
  // Non Blocking I/O
  //
  Status = EfiSptProtocol->PassThru (
                                  EfiSptProtocol,
                                  Target,
                                  Lun,
                                  &Packet,
                                  &Event
                                  );
  . . .

  return Status;
}
```

Example 142. Blocking and Non-blocking Modes

16.7 SCSI Device Enumeration

SCSI device enumeration is performed by platform firmware in UEFI compliant systems. This was the responsibility of the driver in EFI 1.10, but no longer.

The Ext SCSI Pass Thru that is produced by the driver is used by the platform firmware to find all the devices and attach SCSI I/O onto each SCSI device found. It will then try to layer on top of that some other I/O (usually Block I/O) to enable access to the data on the SCSI device.

Note: *The enumeration framework is generic, not only for native SCSI adapters, but also for any SCSI-command-compatible technology such as ATAPI, iSCSI, and Fibre Channel. The example below shows a possible device enumeration framework.*

```
EFI_STATUS
ScsiChannelEnumeration (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath,
  IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL  *ScsiPassThru,
  IN EFI_DEVICE_PATH_PROTOCOL     *ParentDevicePath
  )
{
  EFI_STATUS                  Status;
  UINT32                      ChannelAttribute;
  UINT64                      Lun;
  BOOLEAN                     ScanOtherPuns;
  SCSI_BUS_DEVICE             *ScsiBusDev;
  BOOLEAN                     FromFirstTarget;
  SCSI_TARGET_ID              *ScsiTargetId;
  UINT8                       *TargetId;

  ChannelAttribute = ScsiPassThru->Mode->Attributes;
  if (((ChannelAttribute & EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL) !=
0) &&
      ((ChannelAttribute & EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL)  ==
0) ) {
    //
    // This channel has just PHYSICAL attribute set. Do not do
    // enumeration on this kind of channel.
    //
    return EFI_UNSUPPORTED;
  }


if (RemainingDevicePath == NULL) {
    EfiSetMem (ScsiTargetId, TARGET_MAX_BYTES,0xFF);
    Lun  = 0;
    FromFirstTarget = TRUE;
  } else {
      ScsiBusDev->ExtScsiInterface->GetTargetLun (ScsiBusDev-
>ExtScsiInterface, RemainingDevicePath, &TargetId, &Lun);
    }
  }

  while(ScanOtherPuns) {
    if (FromFirstTarget) {
      //
      // Remaining Device Path is NULL, scan all the possible Puns
      // in the SCSI Channel.
      //
        Status = ScsiBusDev->ExtScsiInterface->GetNextTargetLun
(ScsiBusDev->ExtScsiInterface, &TargetId, &Lun);
      }
      if (EFI_ERROR (Status)) {
        //
        // no legal Pun and Lun found any more
        //
        break;
      }
    } else {
      ScanOtherPuns = FALSE;
    }
    //
    // Avoid creating handle for the host adapter.
    //
      if ((ScsiTargetId->ScsiId.Scsi) == ScsiBusDev->ExtScsiInterface-
>Mode->AdapterId) {
        continue;
      }
    //
    // Scan for the scsi device, if it attaches to the scsi bus,
```

```
    // then create handle and install scsi i/o protocol.
    //
    Status = ScsiScanCreateDevice (This, Controller, ScsiTargetId, Lun,
ScsiBusDev);
  }
  return Status;
}
```

Example 143. SCSI Channel Enumeration

17 Network Driver Design Guidelines

This chapter focuses on the design and implementation of EFI Network Drivers. Most Network controllers are PCI or USB devices, and the Network drivers managing them are also PCI or USB drivers. They must follow all of the design guidelines described in the chapters for that device type, as well as the general guidelines described in chapter 5. In addition, this chapter covers the guidelines that apply specifically to the management of network devices.

See Appendix E of the UEFI specification for Universal Network Driver Interface (UNDI) adapters.

17.1 Overview of Network Drivers

Network drivers are Bus drivers that are used for: add-in network cards, serially attached network cards (USB), cardbus network cards, and LAN-on-mainboard network devices. From the point of view of the upper layers calling into the network stack all of these devices will appear identical.

Network drivers are unique compared to all others peripheral drivers for a few specific reasons.

- They can be runtime drivers. This is not a requirement of a LAN driver, but it is the only class of driver that is documented in the UEFI specification with that option.
- They can have non-protocol interfaces. The Network Interface Identifier Protocol defines the entrypoint to the UNDI structure, but UNDI is not a protocol. Then when a CDB is used it has the abstractions for the driver to use to call back into the hardware.

Note: *These allow an EFI-aware OS to revert to the EFI network stack and download a OS driver.*

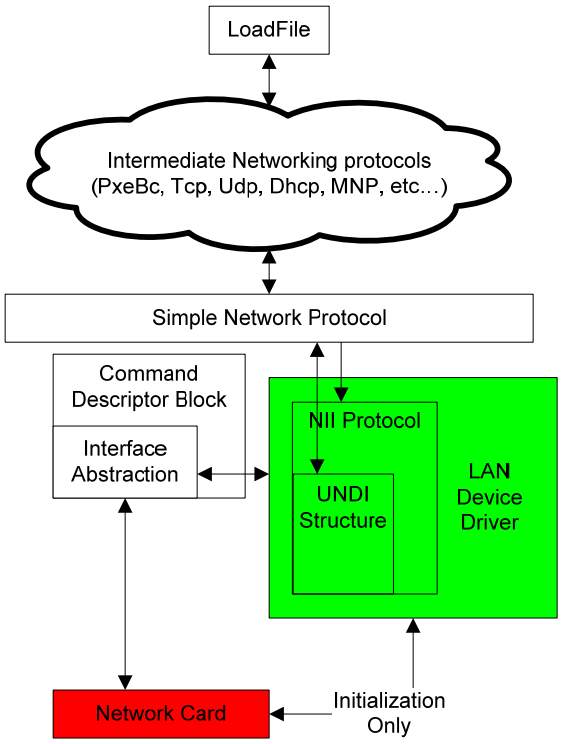

Figure 21. UEFI UNDI Network Stack

Note: *The Command Descriptor Block (CDB) is passed to the UNDI structure by the caller. This includes the interface abstraction.*

17.2 Key Items for Network Drivers

17.2.1 ExitBootServices event

Network drivers need to create an ExitBootServices event (in the DriverBrinding **`Start()`** function) to get notified when the OS Loader is switching to run time operation and shutting down the pre boot environment. Network drivers need to shutdown their receive DMA in response to this event. After ExitBootServices the receive resources allocated to UNDI by SNP will be freed for OS usage. If the network device performs a DMA of received packets into this memory after ExitBootServices it may corrupt OS memory.

17.2.2 SetVirtualAddressMap event

If you have any dynamically allocated memory all those pointers need to be converted to runtime pointers (virtual address pointers). EFI runtime services provide the **`ConvertPointer()`** routine to handle this. This must be done bottom-up since the

virtual pointers are not valid until the SetVirutalAddressMap returns to the OS. The OS is obligated not to call into a runtime EFI driver until the system is in virtual mode.

17.2.3 DriverBinding.Stop()

All protocols opened or exposed by the driver in the **Start()** service should be closed in the reverse order in the **Stop()** service.

17.2.4 Memory Leaks caused by UNDI

When transmitting the UNDI driver needs to keep track of which buffers have been completed, and then return those buffer addresses from the GetStatus API. This allows the top level stack to disposition the buffer (reuse or de-allocate) and not leak memory.

17.3 Implementing Network Driver

There are 2 possible ways to expose the network device to the EFI firmware. This can be done with an UNDI interface exposed through the EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL (NII) which allows a runtime interface. This can also be done by exposing EFI_SIMPLE_NETWORK_PROTOCOL. The EFI_SIMPLE_NETWORK_PROTOCOL would normally be layered on top of the NII by the system firmware, but if it is exposed directly by the device driver the runtime features cannot function.

17.3.1 Network Interface Identifier Protocol

The NII protocol is different from many other protocols in that it has no functions inside it. It is simply a way to share some information with the system and part of that information is the function pointer to the UNDI interface.

Protocol Interface Structure

```
typedef struct {
UINT64  Revision;
UINT64  Id;
UINT64  ImageAddr;
UINT32  ImageSize;
CHAR8   StringId[4];
UINT8   Type;
UINT8   MajorVer;
UINT8   MinorVer;
BOOLEAN Ipv6Supported;
UINT8   IfNum;
} EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL;
```

The piece of this structure that grants access to UNDI is the **Id** field. This field (assuming SW UNDI) contains the address of the beginning of the **!PXE** structure that is the interface to UNDI.

17.3.1.1 UNDI interface

The UNDI interface is made up with the **!PXE** structure.

!PXE SW UNDI				
Offset	0x00	0x01	0x02	0x03
0x00	Signature			
0x04	Len	Fudge	Rev	IFcnt
0x08	Major	Minor	reserved	
0x0C	Implementation			
0x10	Entry Point			
0x14				
0x18	Reserved			#bus
0x1C	Bus Types(s)			
0x20	More Bus Types(s)			

Figure 22. !PXE interface structure.

The EntryPoint is the address of (or offset to) the function that the NII consumer can call into and pass a Command Descriptor Block (CDB).

Command Descriptor Block (CDB)				
Offset	0x00	0x01	0x02	0x03
0x00	OpCode		OpFlags	
0x04	CPBsize		DBsize	
0x08	CPBaddr			
0x0C				
0x10	DBaddr			
0c14				
0x18	StatCode		StatFlags	
0x1C	IFnum		Control	

Figure 23. CDB structure.

The network driver then needs to take the appropriate action with the information. This could be reading or writing to the network, reset, etc. All of the OpCodes are listed in UEFI 2.0 specification section E.3.4.2.

Note: *One of the advantages to the UNDI driver stack is that it is possible to design UNDI capable hardware and use that (there is a separate !PXE HW interface structure) without requiring much of a driver.*

17.3.2 Simple Network Protocol

Exposing SNP instead of NII has some advantages and some disadvantages over using NII. SNP based network drivers are never runtime so the driver does not have to worry about meeting the runtime driver requirements which allows for building in EBC. This may be required for some non-standard network devices that cannot be controlled through a UNDI/NII interface. When a network driver exposes SNP directly the system firmware will simply layer MNP on top of that and will not use its internal SNP driver as part of this network stack.

The following figure shows a possible network stack based on a network driver producing SNP. The inclusion of LoadFile is not guaranteed here, but is a choice made by the system firmware.

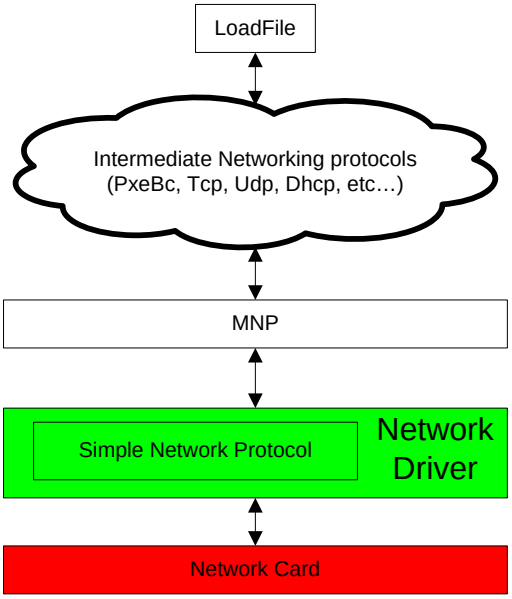

Figure 24. SNP based network stack

17.3.3 Which interface to use

The following table summarizes the features of each of the two possible ways to expose a network interface to the EFI system. The critical 2 that can effect the decision is the Non-UNDI capable device (Obviously if your device does not support UNDI, you have to go with the other) and EBC. There is a significant advantage to compiling for EBC and at this time there is no shipping OS that takes advantage of the runtime capabilities of the UNDI drivers.

Table 30. Network Driver Differences

Feature	NII Driver	SNP Driver	MNP Driver
Can be a Runtime Driver	Yes	No	No

Use Firmware's SNP Driver	Yes	No	No
Use Firmware's MNP Driver	Yes	Yes	No
Requires UNDI capable communication	Yes	No	No
Can be compiled into EBC	No	Yes	Yes
Requires ExitBootServices Event	Yes, if Runtime	No	No
Requires SetVirtualAddressMap event	Yes, if Runtime	No	No
Can talk directly to compliant HW	Yes	No	No

17.3.4 DriverBinding Protocol for network drivers

17.3.4.1 Supported()

The DriverBinding Supported() function is not any different for this device than for any other device. Please refer to the DriverBinding section for details.

17.3.4.2 Start()

The DriverBinding protocol for network drivers is significantly simpler than for many other UEFI Driver Model compliant drivers. All that it does is produce Device Path protocol and whichever one of the two interfaces the driver has chosen as the interface. The following example shows a possible **`Start()`** function for using NII driver stack. There is a complete sample of a NII driver available in the *EDK* at **`\Sample\Bus\Pci\Undi\RuntimeDxe\`**.

There are no known open source samples of a driver that produces SNP directly. There is a SNP driver in the *EDK* that may be of use in developing such drivers. It is located in **`\Sample\Universal\Network\Snp32_64\Dxe\`**.

```
  //
  // Allocate memory
  //
  Status = gBS->AllocatePool (
              EfiRuntimeServicesData,
              (sizeof (PXE_SW_UNDI) + 32),
              &TmpPxePointer
              );
  //
  // Zero the memory
  //
  EfiZeroMem (TmpPxePointer,sizeof (PXE_SW_UNDI) + 32);
  //
  // Verify memory alignment
  //
  if (((UINTN) TmpPxePointer & 0x0F) != 0) {
      pxe_31 = (PXE_SW_UNDI *) ((UINTN) (TmpPxePointer + 8));
    } else {
      pxe_31 = (PXE_SW_UNDI *) TmpPxePointer;
    }
  //
  // Initialize the pxe_31 structure
  //
  PxeStructInit (pxe_31, 0x31);
  //
  // fill NII protocol
  //
  PDS->NIIProtocol.ID = (UINT64) (pxe_31);
  PSD->NIIProtocol.IfNum       = 0;  // assuming only 1 adapter
  PDS->NIIProtocol.Revision       =
EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL_REVISION_31;
  PDS->NIIProtocol.Type           = EfiNetworkInterfaceUndi;
  PDS->NIIProtocol.MajorVer       = PXE_ROMID_MAJORVER;
  PDS->NIIProtocol.MinorVer       = PXE_ROMID_MINORVER_31;
  PDS->NIIProtocol.ImageSize      = 0;
  PDS->NIIProtocol.ImageAddr      = 0;
  PDS->NIIProtocol.Ipv6Supported  = FALSE;

  PDS->NIIProtocol.StringId[0]    = 'U';
  PDS->NIIProtocol.StringId[1]    = 'N';
  PDS->NIIProtocol.StringId[2]    = 'D';
  PDS->NIIProtocol.StringId[3]    = 'I';

  PDS->DeviceHandle                  = NULL;

  //
  // install the NII protocol and Device Path.
  //
  Status = gBS->InstallMultipleProtocolInterfaces (
                 &PDS->DeviceHandle,
                 &gEfiNetworkInterfaceIdentifierProtocolGuid_31,
                 &PDS->NIIProtocol,
                 &gEfiDevicePathProtocolGuid,
                 PDS->Undi32DevPath,
                 NULL
                 );
```

Example 144. Start() with NII protocol version 3.1

17.3.4.3 Stop()

The DriverBinding **Stop()** function does the opposite of the **Start()** function, but it must remember that as a Hybrid driver it has to manage its children also. If there are child handles specified on the parameters then the NII and DevicePath must be

uninstalled on the child and the Io (PciIo, UsbIo, etc…) handle must be closed. If there are no children specified then the Io (PciIo, UsbIo, etc…) and the DevicePath on the controller handle must be closed.

18 Graphics Driver Design Guidelines

This chapter focuses on the design and implementation of EFI Graphics drivers. Most Graphics controllers are PCI controllers, and the Graphics drivers managing them are also PCI drivers. They must follow all of the design guidelines described in chapter 14, as well as the general guidelines described in chapter 5. In addition, this chapter covers the guidelines that apply specifically to the management of Graphics devices.

See section 11.8 of the UEFI 2.0 specification for rules for PCI/AGP graphics adapters.

18.1 Graphics Driver Overview

Please refer to chapter 22 of the *UEFI 2.0 Specification* for details on EBC considerations. Graphics drivers must also create child handles for some of the Graphics output ports and attach **EFI_GRAPHICS_OUTPUT_PROTOCOL** (GOP protocol), **EFI_EDID_DISCOVERED_PROTOCOL** and **EFI_EDID_ACTIVE_PROTOCOL** to each active handle that the driver produced.

The driver can optionally also support **EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL**, but this is not recommended. It is better to let the system firmware produce this on top of the Graphics Output Protocol since the system firmware knows what character sets are required.

18.2 Platform Responsibility

The platform firmware is responsible for determining the role of the graphics driver by changing how the **Start()** function is called, implementing **EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL** on top of GOP, and producing the **EFI_EDID_OVERRIDE_PROTOCOL**.

18.2.1 EDID Override Protocol

The platform firmware may produce **EFI_EDID_OVERRIDE_PROTOCOL** and if it does the graphics driver must take this into account when populating the **EFI_EDID_ACTIVE_PROTOCOL**. The Protocol Interface Structure for shown below has only a single item: the **GetEdid()** function.

Protocol Interface Structure

```
typedef struct _EFI_EDID_OVERRIDE_PROTOCOL EFI_EDID_OVERRIDE_PROTOCOL {
EFI_EDID_OVERRIDE_PROTOCOL_GET_EDID GetEdid;
};
```

The **GetEdid()** function will return the Handle, attributes (override always, never, only if nothing is returned), and the new EDID information. This is then used to initialize the EDID Active Protocol correctly. The function prototype is below.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_EDID_OVERRIDE_PROTOCOL_GET_EDID) (
IN EFI_EDID_OVERRIDE_PROTOCOL     *This,
IN EFI_HANDLE                     *ChildHandle,
   OUT UINT32                     *Attributes,
IN OUT UINTN                      *EdidSize,
IN OUT UINT8                      **Edid
);
```

18.2.2 Simple Text Output Protocol

The platform firmware is required to implement the **EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL** on top of the GOP protocol that is produced by the graphics driver. This allows for optimization in terms of language display requirements; this means that if the firmware is running in mandarin mode, the firmware will know that and load the appropriate UNICODE character set. The graphics driver has no easy way to adjust for this type of issue.

18.3 Implementing Graphics Drivers

EFI Graphics drivers follow the UEFI Driver Model. Depending on the adapter that it manages, a Graphics driver can be categorized as either a single or a multiple output adapter. It must create child handles for each output. Graphics drivers are chip-specific because of the requirement to initialize and manage the Graphics device.

18.3.1 Single Output Graphics Adapters

An example of a Single Output Graphics driver stack is below.

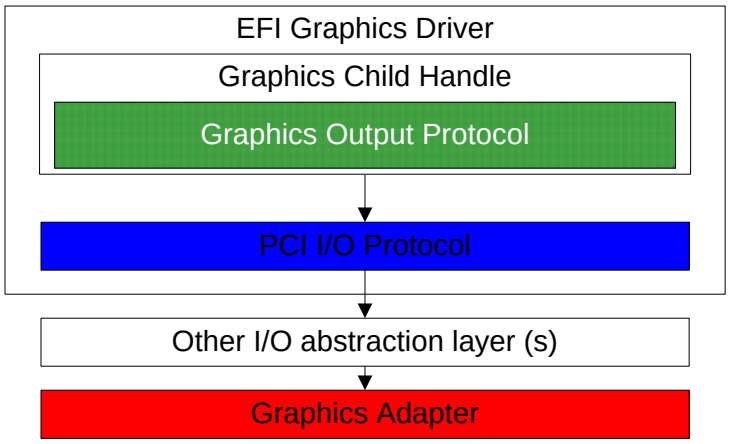

Figure 25. Example Single Output Graphics Driver Implementation

Graphics cards that have a single output device are the simplest type of EFI graphics driver. They produce a single child handle and attach both Device Path and Graphics Output protocols onto that handle. They will need a single data structure to manage the device.

18.3.2 Multiple Output Graphics Adapters

An Example of a Dual Output Graphics Adapter is below.

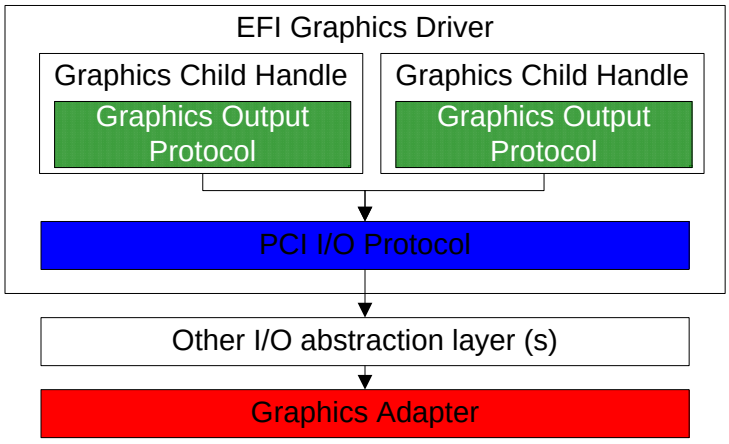

Figure 26. Example Dual Output Graphics Driver Implementation

Multiple output graphics drivers (dual or more) are not significantly more complicated than a single channel adapter in EFI. An important consideration is that many graphics adapters may run in a single output mode in the pre-boot environment; they may then switch to multi-output mode when the higher performance OS driver loads for the device.

18.3.3 Implementing Driver Binding Protocol

Like all drivers that follow the UEFI Driver Model, the image entry point of a Graphics driver installs the Driver Binding Protocol instance on the image handle. All three of the services in the Driver Binding Protocol—**Supported()**, **Start()**, and **Stop()**—must be implemented.

The **Supported()** function tests to see whether the given handle is a manageable adapter. The driver should check that **EFI_DEVICE_PATH_PROTOCOL** and **EFI_PCI_IO_PROTOCOL** are present to ensure the handle that is passed in represents a PCI device. Then the driver should verify that the device is compliant and manageable by reading the *ClassCode*, *VendorId*, and *DeviceId* from the device's PCI configuration header.

The **Start()** function tells the Graphics driver to start managing the controller. In this function, a Graphics driver should use chip-specific knowledge to do the following:

1. Initialize the adapter.
2. Enable the PCI device.
3. Allocate resources.
4. Construct data structures for the driver to use (if required by the device).
5. Enumerate the outputs that are enabled on the device.
6. Create child handles for each detected (and enabled) physical output (physical child handles) and install **EFI_DEVICE_PATH_PROTOCOL**.
7. Get EDID information from each physical output device connected and install **EFI_EDID_DISCOVERED_PROTOCOL** on the child handle.
8. Create child handles for each valid combination of 2 or more video outputs (logical child handles) and install **EFI_DEVICE_PATH_PROTOCOL**.
9. Check **RemainingDevicePath** to see if the correct child or children were created or if **NULL** select a default set. If incorrect children (no defaults) clean up memory and return **EFI_UNSUPPORTED**. If default or correct children set them active.
10. Call **GetEdid()** function to check for overrides on each active physical child handle and produce **EFI_EDID_ACTIVE_PROTOCOL** on each child protocol based on the result.
11. Install **EFI_GRAPHICS_OUTPUT_PROTOCOL** on each active child handle (physical or logical).
12. Install **EFI_COMPONENT_NAME_PROTOCOL**.

The **Start()** function should not scan for devices every time the driver is started. It should depend on **RemainigDevicePath** parameter to determine what to start. Only if **NULL** was passed in should the driver should create a device handle for each device that was found in the scan behind the controller. Otherwise the driver should only start what was specified in **RemainigDevicePath**.

The **Stop()** function performs the opposite operations as **Start()**. Generally speaking, a Graphics driver is required to do the following:

1. Uninstall all protocols on all child handles and close all the child handles.
2. Uninstall all protocols that were attached on the host controller handle.
3. Close all protocol instances that were opened in the Start() function.
4. Release all resources that were allocated for this driver.
5. Disable the adapter.

In general, if it is possible to support RemainigDevicePath, the driver should do so to support the rapid boot capability in the UEFI Driver Model.

18.3.4 Implementing Graphics Output Protocol

The protocol interface for **EFI_GRAPHICS_OUTPUT_PROTOCOL** is listed below:

Protocol Interface Structure

```
typedef struct _EFI_GRAPHICS_OUTPUT_PROTOCOL EFI_GRAPHICS_OUTPUT_PROTOCOL;
struct _EFI_GRAPHICS_OUTPUT_PROTOCOL {
  EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE QueryMode;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE   SetMode;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT        Blt;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE       *Mode;
};
```

The Graphics protocol is a relatively small one in terms of the amount of information. It is also different in that it encapsulates 2 completely separate methods for performing some of the operations. The first method is functional: using the functions to perform operations; the second is direct: using memory pointers to directly access resources.

18.3.4.1 Functional Operations

There are the 3 functions **QueryMode()**, **SetMode()**, and **Blt()** that make up one method. The mode pointer is pointing to a structure that has members so that the consumer of the GOP protocol can get information about the current state.

The **QueryMode()** function is used to return extended information on one of the supported video modes. For example, the protocol consumer could iterate through all of the valid video modes and see what they offer in terms of resolution, color depth, etc. This function will have no effect on the hardware or the currently displayed image.

The **SetMode()** function is how the consumer chooses which video mode they want to be currently active. This is also required to clear the entire display output and reset it all to black.

The **Blt()** function is for transferring information (BLock Transfer) into the video buffer. This is how an image is moved onto the screen. The prototype of the function is below.

```
typedef
EFI_STATUS
(EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT) (
  IN EFI_GRAPHICS_OUTPUT_PROTOCOL      *This,
  IN OUT EFI_GRAPHICS_OUTPUT_BLT_PIXEL *BltBuffer, OPTIONAL
  IN EFI_GRAPHICS_OUTPUT_BLT_OPERATION BltOperation,
  IN UINTN                             SourceX,
  IN UINTN                             SourceY,
  IN UINTN                             DestinationX,
  IN UINTN                             DestinationY,
  IN UINTN                             Width,
  IN UINTN                             Height,
  IN UINTN                             Delta OPTIONAL
  );
```

In this function the driver must translate the entire Blt operation into the correct commands for the graphics adapter that it is managing. This can by be done by performing PCI memory mapped IO or port IO operations or by performing a DMA operation. The exact method is specific to the graphics silicon.

A critical consideration of implementing the **`Blt()`** function is to get the highest performance possible for the user. A common problem is that scrolling the screen results in significant lags such that the user experiences a less than optimal perception. This could be caused by the lags that are normally present when reading back from the frame buffer. A possible solution is to have a copy of the current frame buffer in a memory buffer for use in reads.

The screen is defined in terms of pixels and the buffer is formatted as follows. For a given pixel at location X,Y the location in the buffer is **`Buffer[((Y*<<ScreenWidth>>)+X)]`**. The screen is described according to the following figure.

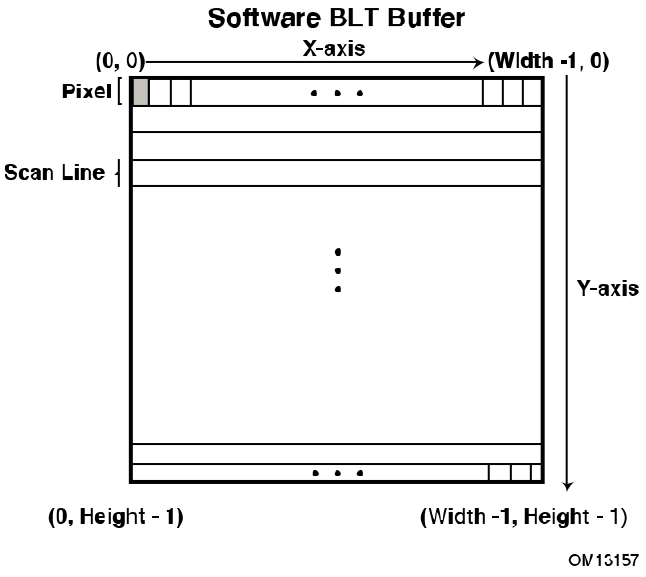

Figure 27. Blt Buffer

An important optimization to make in graphics drivers is for scrolling. Scrolling is one of the most common operations to occur on a pre-boot graphics adapter due to the common use of text base consoles. A method to scroll the screen can be viewed in the *EDK* in the GraphicsConsole driver (**\Sample\Universal\Console\GraphicsConsole\Dxe**).

The **EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE** object pointed to by the **Mode** pointer is populated at the initialization of the adapter, and needs to be updated any time the information changes, for example when the graphics mode changes. The FrameBufferBase member of this object is critical for use by an OS Loader EFI Application in case there is a need to update the screen between the time that the ExitBootServices() function is called and when the OS driver for the Graphics adapter takes over.

18.3.5 Implementing EDID Discovered Protocol

This protocol contains the EDID information that was retrieved from the video output device. This information may differ from the EDID Active Protocol since the EDID Active Protocol will take into account any interaction with the EDID Override Protocol that was consumed by this driver. This protocol must exist on each handle that represents a video output and must only represent a single video output device.

Protocol Interface Structure

```
typedef struct {
  UINT32  SizeOfEdid;
  UINT8   *Edid;
} EFI_EDID_DISCOVERED_PROTOCOL;
```

18.3.6 Implementing EDID Active Protocol

The **EFI_EDID_ACTIVE_PROTOCOL** provides information to the system about a video output device. This will be retrieved from either the **EFI_EDID_DISCOVERED_PROTOCOL** or the **EFI_EDID_OVERRIDE_PROTOCOL**. The protocol interface structure is defined below. The EDID information for the video output device (for example the monitor) connected to this graphics output device is populated into this protocol for use by the system. It is the job of the driver to populate this information if possible. The minimum valid size of EDID information is 128 bytes. See the E-DID EEPROM specification for details on the format of an EDID.

Protocol Interface Structure

```
typedef struct {
  UINT32  SizeOfEdid;
  UINT8   *Edid;
} EFI_EDID_ACTIVE_PROTOCOL;
```

19 Implementing the I/O Protocols

19.1 Block I/O

The EFI_BLOCK_IO_PROTOCOL is used to abstract most mass storage devices to allow upper level applications and drivers access to the contents of the mass storage media without any specialized knowledge of the specific device or controller. This protocol contains functions to read, write, and manage these devices in a generic fashion.

Protocol Interface Structure

```
typedef struct _EFI_BLOCK_IO_PROTOCOL EFI_BLOCK_IO_PROTOCOL {
UINT64              Revision;
EFI_BLOCK_IO_MEDIA  *Media;
EFI_BLOCK_RESET     Reset;
EFI_BLOCK_READ      ReadBlocks;
EFI_BLOCK_WRITE     WriteBlocks;
EFI_BLOCK_FLUSH     FlushBlocks;
};
```

19.1.1 Implementing Reset()

The **Reset()** function resets the block device hardware. During this the driver will also make sure that the device is functioning correctly. Neither of these operations should take a significant amount of time. If the ExtendedVerification flag is set to TRUE, then the driver may take extra time to make sure that the device is functioning. Example 145 shows how to reset an ATAPI device.

```
DeviceSelect = (UINT8) (((bit7 | bit5) | (IdeDev->Device << 4)));
PciIo->Io.Write (
            IdeDev->PciIo,
            EfiPciIoWidthUint8,
            EFI_PCI_IO_PASS_THROUGH_BAR,
            (UINT64) IdeDev->IoPort->Head,
            1,
            &DeviceSelect
            );
Command = ATAPI_SOFT_RESET_CMD;
PciIo->Io.Write (
            IdeDev->PciIo,
            EfiPciIoWidthUint8,
            EFI_PCI_IO_PASS_THROUGH_BAR,
            (UINT64) IdeDev->IoPort->Reg.Command,
            1,
            &Command
            );
// wait for reset to complete…
```

Example 145. Reset() on ATAPI device.

19.1.2 Implementing ReadBlocks() and WriteBlocks()

Reading blocks from many media formats is a similarly defined operation.

1. Verify media presence. This is especially critical for removable or swappable media.

- If there is no media return **EFI_NO_MEDIA**.
- If the media is different return **EFI_MEDIA_CHANGED**

2. Get the media's block size

3. Verify parameters

- The Buffer, sized BufferSize, is a whole number of Blocks
- The read does not start past the end of the media
- The read does not extend past the end of the media
- The buffer is aligned as required

4. read the appropriate sectors from the media

5. copy the appropriate portion of the read into the Buffer

- (Optional) for performance update the driver's cache

This is essentially the same operation as ReadBlocks replacing the read with a write. So for Writing to the media the operation is as follows:

1. Verify media presence. This is especially critical for removable (or swappable) media.

2. If there is no media return **EFI_NO_MEDIA**.

3. If the media is different return **EFI_MEDIA_CHANGED**

4. Get the media's block size
5. Verify parameters
6. The *Buffer*, sized *BufferSize*, is a whole number of Blocks
7. The write does not start past the end of the media
8. The write does not extend past the end of the media
9. The *Buffer* is aligned as required
10. write the appropriate sectors to the media
11. (Optional) for performance update the driver's cache

19.1.3 Implementing FlushBlocks()

FlushBlocks() is used to verify that all pending writes to the disk are completed. This could be used as part of a check before removing some media from the system. This may cause both read and write operations to occur.

19.2 Loadfile

Loadfile is the catch all protocol for any device type which does not fit cleanly into another device type. This should not be used for any standard device type which has a defined driver hierarchy (for example USB, LAN, or SCSI). The protocol interface structure is defined below as from section 12.1 of *UEFI 2.0 specification*.

Protocol Interface Structure

```
typedef struct {
EFI_LOAD_FILE LoadFile;
} EFI_LOAD_FILE_PROTOCOL;
```

The singular function **LoadFile()** of this protocol causes the driver to load the specified file off of it's arbitrary media without the overlying layers knowing anything about the media that the file is stored on.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LOAD_FILE) (
IN EFI_LOAD_FILE_PROTOCOL   *This,
IN EFI_DEVICE_PATH_PROTOCOL *FilePath,
IN BOOLEAN                  BootPolicy,
IN OUT UINTN                *BufferSize,
IN VOID                     *Buffer OPTIONAL
);
```

19.2.1 Implementing LoadFile()

The following steps are done to handle loading files.

1. Verify that the *FilePath* represents a file accessible through this device
2. Verify that *Buffer* is large enough by examining *BufferSize* parameter
3. If not large enough, place correct size in *BufferSize* and return **EFI_BUFFER_TOO_SMALL**
4. See if file Exists
5. If no, check *BootPolicy* to see if inexact *FilePath* may be present
6. If PXE boot is occurring and *BootPolicy* is **FALSE** follow special steps.

19.3 Console Protocols

19.3.1 Simple Text In

Simple Text In is a protocol produced by anything that can act like a basic keyboard. This could be an actual keyboard, whether USB, PS/2, or wireless or a simulated keyboard like a Telnet server.

Protocol Interface Structure

```
typedef struct _EFI_SIMPLE_TEXT_INPUT_PROTOCOL
EFI_SIMPLE_TEXT_INPUT_PROTOCOL {
EFI_INPUT_RESET    Reset;
EFI_INPUT_READ_KEY ReadKeyStroke;
EFI_EVENT          WaitForKey;
};
```

19.3.1.1 Implementing EFI_SIMPLE_TEXT_INPUT_PROTOCOL.Reset()

The reset function is for resetting the input device hardware. This only takes a single parameter which is whether to do an extended or a basic functionality test following the reset operation. This functions implementation is dependant on the underlying hardware specifications.

19.3.1.2 Implementing EFI_SIMPLE_TEXT_INPUT_PROTOCOL.ReadKeyStroke()

The **ReadKeyStroke()** function is a non-blocking operation that will return immediately with its **Key* parameter containing the keycode for the next key in the queue or it will return that there was no keycode ready. It will never wait for a key to be pressed.

19.3.1.3 Using EFI_SIMPLE_TEXT_INPUT_PROTOCOL.WaitForKey

WaitForKey is of type **EFI_EVENT** that can be waited upon with the **WaitForEvent()** function. It is the responsibility of the producing driver to signal this event when a key code becomes ready for reading.

19.3.2 Simple Text Out

Simple Text Out is produced by drivers which can act like a basic visual interface. Due to the limitation to this device to support text only and no graphics this is used primarily by Telnet servers and low end graphics cards. This protocol is also produced by platform firmware on top of any device that has produced Graphics Output Protocol.

Protocol Interface Structure

```
typedef struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
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;
} EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL;
```

19.3.2.1 Implementing EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.Reset()

The reset here can be as simple as resetting the mode and clearing the screen, as demonstrated by Example 146 which is from the Terminal driver located at **\Sample\Universal\Console\Terminal\Dxe** in the *EDK*.

```
This->SetAttribute (This,
                    EFI_TEXT_ATTR (This->Mode->Attribute & 0x0F,
                    EFI_BACKGROUND_BLACK)
                    );
Status = This->SetMode (This, 0);
```

Example 146. light reset of terminal driver

It can also easily perform more actions. When the ExtendedVerification parameter is TRUE this same driver will also reset the serial protocol that it is running on top of. See Example 147.

```
if (ExtendedVerification) {
  //
  // Report progress code here
  //
  ReportStatusCodeWithDevicePath (
    EFI_PROGRESS_CODE,
    EFI_PERIPHERAL_REMOTE_CONSOLE | EFI_P_PC_RESET,
    0,
    &gTerminalDriverGuid,
    TerminalDevice->DevicePath
    );
  Status = TerminalDevice->SerialIo->Reset (TerminalDevice->SerialIo);
  if (EFI_ERROR (Status)) {
    //
    // Report error code here
    //
    ReportStatusCodeWithDevicePath (
      EFI_ERROR_CODE | EFI_ERROR_MINOR,
      EFI_PERIPHERAL_REMOTE_CONSOLE | EFI_P_EC_CONTROLLER_ERROR,
      0,
      &gTerminalDriverGuid,
      TerminalDevice->DevicePath
      );
    return Status;
  }
}
```

Example 147. Full reset of Terminal driver

19.3.2.2 Implementing EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.OutputString()

OutputString() is the function used to output Unicode strings to the console. It is responsible for verifying the printability of the string passed, fixing it if required, and displaying it on the console. The steps to follow are:

1. Verify that the current mode is good
2. Verify that each character is printable as text or graphics
3. Fix any that are not printable
4. Print all characters
5. Update position of the cursor in the *Mode
6. Return **EFI_SUCCESS** or **EFI_WARN_UNKNOWN_GLYPH** if some had to be fixed before printing

19.3.2.3 Implementing EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.TestString()

The **TestString()** function verifies that all the characters in the string can be printed. That is that they will not need to be fixed if they were passed into the **OutputString()** function. Using the same internal function to do the verification for the two functions is a good way to make sure that these functions are consistent.

19.3.2.4 Implementing EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.QueryMode()

The **QueryMode()** function is normally used one of two ways:

1. Query for the geometry of the current mode. The following line will populate the Column and Row variables with the geometry of the currently active console output.

```
This->QueryMode(This, This->Mode->Mode, &Column, &Row)
```

2. Loop through all valid geometries that a given console can support. The following line will populate (repeatedly) the Column and Row variables with the geometry of the each supported output mode.

```
for (LoopVariable = 0 ; LoopVariable < This->Mode->MaxMode ;
++LoopVariable)
  This->QueryMode(This, LoopVariable, &Column, &Row);
```

The driver's job is to populate this information correctly for any value of *ModeNumber* which is less than **Mode->MaxMode** and greater than or equal to zero.

19.3.2.5 Implementing EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.SetMode()

The **SetMode()** function is used to select which of the supported output modes the upper layer wishes to use. The choice should be verified to be a supportable mode and then the selected mode should be made the currently active output mode. After this done (and success is guaranteed) update the **Mode->Mode** variable with the new currently active mode.

19.3.2.6 Implementing EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.SetAttribute()

Setting the attributes is how the upper layers define how the screen printing should occur. This will affect the background and foreground colors that are used when either **OutputString()** or **ClearScreen()** is called. This function by itself will not change anything already printed to the console.

19.3.2.7 Implementing EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.ClearScreen()

ClearScreen() makes the entire console have no text on it and makes it all the currently selected background color. The cursor is also set to the (0, 0) position (upper left square).

19.3.2.8 Implementing EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.SetCursorPosition()

SetCursorPosition() selects a new location for the cursor within the currently selected console's valid geometry. The new position's row must be less than the Row returned to **QueryMode()** and likewise the new position's column must be less than the

Column returned to **QueryMode()**. See Figure 27 for a representation of the screen coordinates.

19.3.2.9 Implementing EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.EnableCursor()

The **EnableCursor()** function tells the driver whether or not to show the cursor on the screen. This has no impact on the position or functionality of the cursor, but only its visible state.

19.3.3 Serial I/O

Serial I/O is produced by any device that has a UART style serial port on it. This would be produced by serial adapters (add-in card or USB), any integrated serial ports off the main system, and other character based I/O devices.

Note: Mode *needs to be updated each time that* ***SetControl()*** *or* ***SetAttributes()*** *is successful. This makes the upper layers able to get access to the information about the serial device.*

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;
} EFI_SERIAL_IO_PROTOCOL;
```

19.3.3.1 Implementing EFI_SERIAL_IO_PROTOCOL.Reset()

When this function gets called by the protocol consumer is the responsibility of the driver to reset the hardware. There is no basic or extended functionality required for this reset function unlike the other reset functions in the console protocols.

19.3.3.2 Implementing EFI_SERIAL_IO_PROTOCOL.SetAttributes()

The **SetAttributes()** function is used by the caller to change the serial connection's attributes for *BaudRate*, *ReceiveFifoDepth*, *Timeout*, *Parity*, *DataBits*, and *StopBits*. The caller will pass in 0 for any of these values that should be set to the default value. *Parity* and *StopBits* are enumerated values with their default value set in the 0th.

If any of the parameters is an invalid value then the function will return **EFI_INVALID_PARAMETER**; the only other valid fail return value is **EFI_DEVICE_ERROR** if the serial device is physically not functioning correctly.

The *Mode* pointer should be updated in this function when success has been determined, but not modified if there is an error.

19.3.3.3 Implementing EFI_SERIAL_IO_PROTOCOL.SetControl() and GetControl()

GetControl() and **SetControl()** are used to view and modify respectively the control bits on the serial device. All of the values listed Table 31 can be read back with **GetControl()**, but some can not be modified with **SetControl()**. If a non-modifiable bit is attempted to be set with **SetControl()** the driver should return an error.

The *Mode* pointer should be updated in this function when success has been determined, but not modified if there is an error.

Table 31. Serial I/O Protocol control bits

Control Bit #define	Modifiable with SetControl()
EFI_SERIAL_CLEAR_TO_SEND	NO
EFI_SERIAL_DATA_SET_READY	NO
EFI_SERIAL_RING_INDICATE	NO
EFI_SERIAL_CARRIER_DETECT	NO
EFI_SERIAL_REQUEST_TO_SEND	YES
EFI_SERIAL_DATA_TERMINAL_READY	YES
EFI_SERIAL_INPUT_BUFFER_EMPTY	NO
EFI_SERIAL_OUTPUT_BUFFER_EMPTY	NO
EFI_SERIAL_HARDWARE_LOOPBACK_ENABLE	YES
EFI_SERIAL_SOFTWARE_LOOPBACK_ENABLE	YES
EFI_SERIAL_HARDWARE_FLOW_CONTROL_ENABLE	YES

19.3.3.4 Implementing EFI_SERIAL_IO_PROTOCOL.Write() and Read()

The **Write()** and **Read()** functions are used to write bytes out to the serial device or read in from the serial device. The only 2 parameters that are passed are the number of bytes and then either the buffer to write out or a buffer to read the bytes into. The amount of time that this can take is determined by the Timeout value in the *Mode* object (as set by **SetAttributes()**).

19.3.4 Graphics Output Protocol

Please see section 18.3.4 for information on GOP protocol.

20 Driver Optimization Techniques

There are several techniques that can be used to optimize an UEFI driver. These techniques can be broken down into the following two categories:

- Techniques to reduce the size of UEFI drivers
- Techniques to improve the performance of UEFI drivers

Sometimes these techniques complement each other, and sometimes they are at odds with each other. For example, an UEFI driver may grow in size to meet a specific performance goal. The driver writer will have to make the appropriate compromises in the selection of these driver optimization techniques.

20.1 Space Optimizations

Table 32 lists the techniques that can be used to reduce the size of UEFI drivers. By using combinations of all of these techniques, significant size reductions can be realized. The compiler and linker switches that are referenced below are specific to the Microsoft Visual Studio tool chain. Different compilers and linkers may use different switches for equivalent operations.

Table 32. Space Optimizations

Technique	Description
UEFI driver Library	The UEFI driver Library should be used to reduce the size of UEFI drivers. It is a lightweight library that contains only the functions that most UEFI drivers require. The UEFI driver Library consists of three different groups of functions: General-purpose library functions String library functions Print library functions Most UEFI drivers use the general-purpose library. UEFI drivers that need to manipulate Unicode and ASCII strings use the string library, and UEFI drivers that require the ability to generate formatted strings use the print library. Any UEFI driver that uses the **`DEBUG()`** macro will also require the print library.

/Os or /O1 Compiler Switches	Both the /Os and /O1 compiler switches will optimize a C compiler for size. This is an easy way to reduce the size of an UEFI driver significantly. Care must be taken when turning on compiler optimizations because C source that works fine with optimizations disabled may stop working with optimizations enabled. They usually stop working because of missing **`volatile`** declarations on variables and data structures that are shared between normal contexts and raised TPL contexts. Also, because the UEFI driver is small, it may execute faster. If there are any speed paths in an UEFI driver that will cause problems if the UEFI driver executes faster, then these switches may expose those speed paths. These same speed paths will also show up as faster processors are used, so it is good to find these speed paths early.
/OPT:REF Linker Switch	This linker switch removes unused functions and variables from the executable image, including functions and variables in the UEFI driver and the libraries against which the UEFI driver is being linked. The combination of using the UEFI driver Library with this linker switch can significantly reduce the size of an UEFI driver executable. Also, **`DEBUG()`** macros are removed when a production build is performed, so using this linker switch will remove the print library from the executable image.
EFI Compression	If an UEFI driver is going to be stored in a PCI option ROM, then the EFI compression algorithm can be used to reduce the size of an UEFI driver further. The build utility **`EfiRom`** has built-in support for compressing UEFI images, and the PCI bus driver has built-in support for decompressing UEFI drivers stored in PCI option ROMs. The average compression ratio on IA-32 is 2.3, and the average compression ratio on the Itanium processor is 2.8. The EfiRom utility is described in chapter 18.
EFI Byte Code Images	If an UEFI driver is going to be stored in a PCI option ROM and the PCI option ROM must support both IA-32 and Itanium-based platforms or just Itanium-based platforms, then EFI Byte Code (EBC) executables should be considered. EBC executables are portable between IA-32 and Itanium processors. This portability means that only a single UEFI driver image is required to support both IA-32 and Itanium-based platforms. Also, the EBC executables are significantly smaller than images for the Itanium processor, so there are advantages to using this format for UEFI drivers that are targeted only at Itanium-based platforms. In addition, using *EFI Compression* (see above) can reduce the EBC executables even further.

20.2 Speed Optimizations

Table 33 lists the techniques that can be used to improve the performance of UEFI drivers. By using combinations of all of these techniques, significant performance enhancements can be realized.

Table 33. Speed Optimizations

Technique	Description
/Ot or /O1 Compiler Switches	The /Ot switch optimizes for code speed, and the /O1 compiler switch optimizes a C compiler for size. This technique is an easy way to reduce the execution time and significantly reduce the size of an UEFI driver. Care must be taken when turning on compiler optimizations because C source that works fine with optimizations disabled may stop working with optimizations enabled. They usually stop working because of missing **volatile** declarations on variables and data structures that are shared between normal contexts and raised TPL contexts. Also, because the UEFI driver is small, it may execute faster. If there are any speed paths in an UEFI driver that will cause problems if the UEFI driver executes faster, then these switches may expose those speed paths. These same speed paths will also show up as faster processors are used, so it is good to find these speed paths early.
/O2 Compiler Switch	The /O2 compiler switch will optimize a C compiler for speed. This technique is an easy way to improve the performance of an UEFI driver. Care must be taken when turning on compiler optimizations because C source that works fine with optimizations disabled may stop working with optimizations enabled. They usually stop working because of missing **volatile** declarations on variables and data structures that are shared between normal contexts and raised TPL contexts. If there are any speed paths in an UEFI driver that will cause problems if the UEFI driver executes faster, then this switch may expose those speed paths. These same speed paths will also show up as faster processors are used, so it is good to find these speed paths early. The use of the /O2 compiler switch may actually increase the size of an UEFI driver.
EFI Services	Whenever possible, use EFI Boot Services, EFI Runtime Services, and the protocol services provided by other UEFI drivers. The EFI Boot Services and EFI Runtime Services will likely be native calls that have been optimized for the platform, so there will always be a performance advantage for using these services. Some protocol services might be native, and other protocol services might be EBC images. Either way, if all UEFI drivers assume that external protocol services are native, then the combination of UEFI drivers and EFI services will result in more efficient execution.
PCI I/O Protocol	If an UEFI driver is a PCI driver, then it should take advantage of all the PCI I/O Protocol services to improve the UEFI driver's performance. This approach means that all register accesses should be performed at the largest possible size. For example, perform a single 32-bit read instead of multiple 8-bit reads. Also, take advantage of the read/write multiple, FIFO, and fill modes of the **Io()**, **Mem()**, and **Pci()** services.

20.2.1 CopyMem() and SetMem() Operations

Example 148 shows examples of how **gBS->CopyMem()** and **gBS->SetMem()** should be used to improve the performance of an UEFI driver. These techniques apply to arrays, structures, or allocated buffers.

```
typedef struct {
  UINT8   First;
  UINT32  Second;
} MY_STRUCTURE;

UINTN         Index;
UINT8         A[100];
UINT8         B[100];
MY_STRUCTURE  MyStructureA;
MY_STRUCTURE  MyStructureB;

//
// Using a loop is slow or structure assignments is slow
//
for (Index = 0; Index < 100; Index++) {
  A[Index] = B[Index];
}
MyStructueA = MyStructureB;

//
// Using the optimized CopyMem() Boot Services is fast
//

gBS->CopyMem((VOID *)A, (VOID *)B, 100);
gBS->CopyMem((VOID *)MyStructureA, (VOID *)MyStructureB, sizeof
(MY_STRUCTURE));

//
// Using a loop or individual assignment statements is slow
//
for (Index = 0; Index < 100; Index++) {
  A[Index] = 0;
}
MyStructureA.First = 0;
MyStructureA.Second = 0;

//
// Using the optimized SetMem() Boot Service is fast.
//
gBS->SetMem((VOID *)A, 100, 0);
gBS->SetMem((VOID *)&MyStructureA, sizeof (MY_STRUCTURE), 0);
```

Example 148. CopyMem() and SetMem() Speed Optimizations

20.2.2 PCI I/O Fill Operations

The following Examples show ways to fill video frame buffer with zeros on a PCI video controller. The frame buffer is 1 MB of memory-mapped I/O that is accessed through BAR #0 of the PCI video controller. The following four examples of performing this operation are shown, from slowest to fastest:

Two methods that can significantly increase the performance of an UEFI driver are taking advantage of the fill operations to eliminate loops and writing to a PCI controller at the largest possible size.

```
EFI_PCI_IO_PROTOCOL  *PciIo;
UINT32               Color32;
UINTN                Count;
UINTN                Index;

//
// This is the slowest method. It performs 0x100000 calls through PCI I/O
and
// writes to the frame buffer 8 bits at a time.
//
Color8 = 0;
for (Index = 0; Index < 0x100000; Index++) {
  Status = PciIo->Mem.Write (
                       PciIo,
                       EfiPciIoWidthUint8,        // Width
                       0,                         // Bar Index
                       Index,                     // Offset
                       1,                         // Count
                       &Color8                    // Value
                       );
}
```

Example 149. Uses a loop to write to the frame buffer 8 bits at a time.

```
//
// This is a little better. It performs 0x100000/4 calls through PCI I/O
and
// writes to the frame buffer 32 bits at a time.
//
Color32 = 0;
for (Index = 0; Index < 0x100000; Index += 4) {
  Status = PciIo->Mem.Write (
                       PciIo,
                       EfiPciIoWidthUint32,       // Width
                       0,                         // Bar Index
                       Index,                     // Offset
                       1,                         // Count
                       &Color32                   // Value
                       );
}
```

Example 150. Uses a loop to write to the frame buffer 32 bits at a time.

```
//
// This is much better. It performs 1 call to PCI I/O, but it is writing
the
// frame buffer 8 bits at a time.
//
Color8 = 0;
Count = 0x100000;
Status = PciIo->Mem.Write (
                     PciIo,
                     EfiPciIoWidthFillUint8,    // Width
                     0,                         // Bar Index
                     0,                         // Offset
                     Count,                     // Count
                     &Color8                    // Value
                     );
```

Example 151. Does not use a loop and writes 8 bits at a time to frame buffer.

```
//
// This is the best method. It performs 1 call to PCI I/O, and it is
writing
// the frame buffer 32 bits at a time.
//
Color32 = 0;
Count = 0x100000 / sizeof (UINT32);
Status = PciIo->Mem.Write (
                     PciIo,
                     EfiPciIoWidthFillUint32,   // Width
                     0,                         // Bar Index
                     0,                         // Offset
                     Count,                     // Count
                     &Color32                   // Value
                     );
```

Example 152. Does not use a loop and writes 32 bits at a time to frame buffer.

20.2.3 PCI I/O FIFO Operations

Example 153 shows an example of writing a sector to an IDE controller. The IDE controller uses a single 16-bit I/O port as a FIFO for reading and writing sector data. The first example calls the PCI I/O Protocol 256 times to write the sector. The second example is much better, because it calls the PCI I/O Protocol only once, which will provide a significant performance increase if this example is compiled with an EBC compiler. This example would apply equally to FIFO read operations.

```
EFI_PCI_IO_PROTOCOL  *PciIo;
UINT16               Buffer[256];
UINTN                Index;

//
// This is the slowest method. It performs 256 PCI I/O calls to write 256
// 16-bit values to the IDE controller.
//
for (Index = 0; Index < 256; Index++) {
  Status = PciIo->Io.Write (
                      PciIo,
                      EfiPciIoWidthUint16,
                      EFI_PCI_IO_PASS_THROUGH_BAR,
                      0x1F0,
                      1,
                      Buffer[Index]
                      );
}

//
// This is the fastest method. It uses a loop to write 256 16-bit values
to
// the IDE controller.
//
Status = PciIo->Io.Write (
                     PciIo,
                     EfiPciIoWidthFifoUint16,
                     EFI_PCI_IO_PASS_THROUGH_BAR,
                     0x1F0,
                     256,
                     Buffer
                     );
```

Example 153. Speed Optimizations Using PCI I/O FIFO Operations

20.2.4 PCI I/O CopyMem() Operations

The following examples show how scrolling a frame buffer by different methods can change performance. In the first it is done one scan line at a time.. Just like the **gBS->CopyMem()** Boot Service, the **CopyMem()** service in the PCI I/O Protocol should be used whenever it can eliminate loops. The frame buffer is 1 MB of memory-mapped I/O that is accessed through BAR #0 of the PCI video controller, and the screen is 800 pixels wide with 32 bits per pixel.In the second it is done with a single call to generate the exact same result. This second example would run significantly faster if it was compiled with an EBC compiler.

```
EFI_PCI_IO_PROTOCOL  *PciIo;
UINTN                Index;
UINT32               Value;
UINTN                ScanLineWidth;

//
// This is the slowest method. It performs almost 0x100000/4 read
// and write accesses through the PCI I/O protocol.
//
ScanLineWidth = 800 * 4;
for (Index = ScanLineWidth; Index < 0x100000; Index += 4) {
  Status = PciIo->Mem.Read (
                        PciIo,
                        EfiPciIoWidthUint32,    // Width
                        0,                      // Bar Index
                        Index,                  // Offset
                        1,                      // Count
                        &Value                  // Value
                        );
  Status = PciIo->Mem.Write (
                        PciIo,
                        EfiPciIoWidthUint32,    // Width
                        0,                      // Bar Index
                        Index - ScanLineWidth,  // Offset
                        1,                      // Count
                        &Value                  // Value
                        );
}
```

Example 154. Scrolling the screen through a loop

```
//
// This is the faster method. It makes a single call to CopyMem().
//
Status = PciIo->CopyMem (
                  PciIo,
                  EfiPciIoWidthUint32,    // Width
                  0,                      // Destination Bar Index
                  0,                      // Destination Offset
                  0,                      // Source Bar Index
                  ScanLineWidth,          // Source Offset
                  0x100000/4              // Count
                  );
```

Example 155. Scrolling the screen without a loop

20.2.5 PCI Configuration Header Operations

The following 3 examples demonstrate that reading the PCI configuration header from a PCI controller can be dramatically affected by the method chosen. In the first

example a loop is used to read the header 8 bits at a time; in the second example a single call is used to read the entire header 8 bits at a time; in the third example a single call is used to read the header 32 bits at a time.

```
EFI_PCI_IO_PROTOCOL  *PciIo;
PCI_TYPE00           Pci;
UINTN                Index;

//
// Loop reading the 64-byte PCI configuration header 8 bits at a time
//
for (Index = 0; Index < sizeof (Pci); Index++) {
  Status = PciIo->Pci.Read (
                      PciIo,
                      EfiPciIoWidthUint8,      // Width
                      Index,                   // Offset
                      1,                       // Count
                      (UINT8 *)(&Pci) + Index  // Value
                      );
}
```

Example 156. reading PCI configuration with a loop 8 bits at a time

```
//
// This is a faster method that removes the loop and reads 8 bits at a
time.
//
Status = PciIo->Pci.Read (
                    PciIo,
                    EfiPciIoWidthUint8,      // Width
                    0,                       // Offset
                    sizeof (Pci),            // Count
                    &Pci                     // Value
                    );
```

Example 157. reading PCI configuration without a loop 8 bits at a time

```
//
// This is the fastest method that makes a single call to PCI I/O and
reads the
// PCI configuration header 32 bits at a time.
//
Status = PciIo->Pci.Read (
                    PciIo,
                    EfiPciIoWidthUint32,               // Width
                    0,                                 // Offset
                    sizeof (Pci) / sizeof (UINT32),    // Count
                    &Pci                               // Value
                    );
```

Example 158. reading PCI configuration without a loop 32 bits at a time

20.2.6 PCI I/O Read/Write Multiple Operations

The following examples demonstrate how writing to a PCI memory-mapped I/O buffer can dramatically affect the performance of a UEFI driver. In the first example a loop is used with 8 bit operations, while in the second the same operation is done with a single call. This example is based on writing to a 1MB frame buffer by a video card driver.

Note: *The examples show here apply equally well to reading a bitmap from the frame buffer of a PCI video controller using the* ***`PciIo->Mem.Read()`*** *function.*

```
EFI_PCI_IO_PROTOCOL  *PciIo;
UINTN                Index;
UINT8                Bitmap[0x100000];

//
// Loop writing a 1 MB bitmap to the frame buffer 8 bits at a time.
//
for (Index = 0; Index < sizeof (BitMap); Index++) {
  Status = PciIo->Mem.Write (
                      PciIo,
                      EfiPciIoWidthUint8,      // Width
                      0,                       // BarIndex
                      Index,                   // Offset
                      1,                       // Count
                      &BitMap[Index]           // Value
                      );
}
```

Example 159. Writing 1MB frame buffer with a loop 8 bits at a time

```
//
// This is a faster method that removes the loop and writes 32 bits at a
time.
//
Status = PciIo->Mem.Write (
                    PciIo,
                    EfiPciIoWidthUint32,              // Width
                    0,                               // BarIndex
                    0,                                // Offset
                    sizeof (BitMap) / sizeof (UINT32),// Count
                    BitMap                            // Value
                    );
```

Example 160. Writing a 1MB frame buffer with a single call

20.2.7 PCI I/O Polling Operations

These same types of optimization can be applied to polling also. In the following examples the different methods shown for polling will be: a loop with 10 µS stalls to wait up to 1 minute and a single call to PCI I/O protocol to perform the entire operation. This type of poll is usually done when a driver is waiting for the hardware to complete an operation, and the completion status is indicated by a bit changing state in an I/O port or a memory-mapped I/O port. In the examples below poll offset 0x20 in BAR #1 for bit 0 to change from a 0 to a 1.

The **PollIo()** and **PollMem()** functions in the PCI I/O Protocol are very flexible, and they can simplify the operation of polling for bits to change state in status registers.

```
EFI_PCI_IO_PROTOCOL  *PciIo;
UINTN                TimeOut;
UINT8                Result8;

//
// Loop for up to 1 second waiting for Bit #0 in register 0x20 of BAR #1
to
// become a 1.
//
TimeOut = 0;
do {
  Status = PciIo->Mem.Read (
                        PciIo,
                        EfiPciIoWidthUint8,      // Width
                        1,                       // BarIndex
                        0x20,                    // Offset
                        1,                       // Count
                        &Result8                 // Value
                        );
  if ((Result8 & 0x01) == 0x01) {
    return EFI_SUCCESS;
  }
  gBS->Stall (10);
  TimeOut = TimeOut + 10;
} while (TimeOut < 1000000);
return EFI_TIMEOUT;
```

Example 161. Polling for 1 second through a loop

```
UINT64               Result64;
//
// Call PollIo() to poll for Bit #0 in register 0x20 of Bar #1 to be set
to a 1.
//
Status = PciIo->Pci.PollIo (
                      PciIo,
                      EfiPciIoWidthUint8,      // Width
                      1,                       // BarIndex
                      0x20,                    // Offset
                      0x01,                    // Mask
                      0x01,                    // Value
                      10000000,                // Poll for 1 second
                      &Result64                // Result
                      );
```

Example 162. Polling for 1 second using polling function

20.3 Compliance Optimization

Complying with different versions of EFI specifications is critical for many drivers. If the driver is required to work on current EFI 1.10 (like most Itanium) systems and also on next generation UEFI 2.0 systems than the driver needs to be optimized for compliance.

There are 2 parts to this: protocol consumption and protocol production.

20.3.1 Protocol Production

Produce any protocol used in either version of the specification. For example a USB HC driver would product both the **EFI_USB_HC_PROTOCOL** and the **EFI_USB2_HC_PROTOCOL**

or a SCSI driver would produce both **EFI_SCSI_PASSTHRU_PROTOCOL** and **EFI_EXT_SCSI_PASSTHRU_PROTOCOL**. This allows for the driver to function under a firmware that is compliant to any of the versions of EFI specifications. Example 163 shows code from the UNDI driver in the *EDK*.

```
  UNDI32Device->Signature                      = UNDI_DEV_SIGNATURE;

  UNDI32Device->NIIProtocol.Revision           =
EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL_REVISION;
  UNDI32Device->NIIProtocol.Type               = EfiNetworkInterfaceUndi;
  UNDI32Device->NIIProtocol.MajorVer           = PXE_ROMID_MAJORVER;
  UNDI32Device->NIIProtocol.MinorVer           = PXE_ROMID_MINORVER;
  UNDI32Device->NIIProtocol.ImageSize          = 0;
  UNDI32Device->NIIProtocol.ImageAddr          = 0;
  UNDI32Device->NIIProtocol.Ipv6Supported      = FALSE;

  UNDI32Device->NIIProtocol.StringId[0]        = 'U';
  UNDI32Device->NIIProtocol.StringId[1]        = 'N';
  UNDI32Device->NIIProtocol.StringId[2]        = 'D';
  UNDI32Device->NIIProtocol.StringId[3]        = 'I';

  UNDI32Device->NIIProtocol_31.Revision        =
EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL_REVISION_31;
  UNDI32Device->NIIProtocol_31.Type            = EfiNetworkInterfaceUndi;
  UNDI32Device->NIIProtocol_31.MajorVer        = PXE_ROMID_MAJORVER;
  UNDI32Device->NIIProtocol_31.MinorVer        = PXE_ROMID_MINORVER_31;
  UNDI32Device->NIIProtocol_31.ImageSize       = 0;
  UNDI32Device->NIIProtocol_31.ImageAddr       = 0;
  UNDI32Device->NIIProtocol_31.Ipv6Supported   = FALSE;

  UNDI32Device->NIIProtocol_31.StringId[0]     = 'U';
  UNDI32Device->NIIProtocol_31.StringId[1]     = 'N';
  UNDI32Device->NIIProtocol_31.StringId[2]     = 'D';
  UNDI32Device->NIIProtocol_31.StringId[3]     = 'I';

  UNDI32Device->DeviceHandle                   = NULL;

  //
  // install both the 3.0 and 3.1 NII protocols.
  //
  Status = gBS->InstallMultipleProtocolInterfaces (
                 &UNDI32Device->DeviceHandle,
                 &gEfiNetworkInterfaceIdentifierProtocolGuid_31,
                 &UNDI32Device->NIIProtocol_31,
                 &gEfiNetworkInterfaceIdentifierProtocolGuid,
                 &UNDI32Device->NIIProtocol,
                 &gEfiDevicePathProtocolGuid,
                 UNDI32Device->Undi32DevPath,
                 NULL
                 );
```

Example 163 Producing both NII versions

20.3.2 Protocol Consumption

Consume both versions of a protocol in precedence order. For example network drivers will search for NII 3.1 and then later search for NII 3.0 protocol. This means that if NII 3.1 is found it is used, but if it is not found NII 3.0 will be used instead (with a reduces feature set). See Example 164 for a sample.

```
//
// Get the NII interface
//
Status = gBS->OpenProtocol (
                Controller,
                &gEfiNetworkInterfaceIdentifierProtocolGuid_31,
                &Private->NiiPtr,
                This->DriverBindingHandle,
                Controller,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );

if (EFI_ERROR (Status)) {
  Status = gBS->OpenProtocol (
                  Controller,
                  &gEfiNetworkInterfaceIdentifierProtocolGuid,
                  &Private->NiiPtr,
                  This->DriverBindingHandle,
                  Controller,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );

  if (EFI_ERROR (Status)) {
    goto PxeBcError;
  }
}
```

Example 164 Searching for newer and then older versions of a protocol.

21 Itanium Architecture Porting Considerations

When writing an UEFI driver, there are several steps that can be taken to ensure that the driver will function properly on an Itanium-based platform. Typically, UEFI drivers are initially developed for an IA-32 platform and then will be ported to an Itanium-based platform. If an UEFI driver contains IA-32 assembly language sources, then those sources must be converted to C or assembly language sources for the Itanium processor. In general, it is always better to write UEFI drivers in C so the driver will be as portable as possible. The guidelines listed in this chapter will help improve the portability of an UEFI driver and it specifically addresses the pitfalls that may be encountered when an UEFI driver is ported to an Itanium processor.

21.1 Alignment Faults

The single largest issue with UEFI drivers for Itanium-based platforms is alignment. An IA-32 processor will allow any sized transaction on any byte boundary. The Itanium processor allows transactions to be performed only on natural boundaries. This requirement means that a 64-bit read or write transaction must begin on an 8-byte boundary, a 32-bit read or write transaction must begin on a 4-byte boundary, and a 16-bit read or write transaction must begin on a 2-byte boundary. In most cases, the driver writer does not need to worry about this issue because the C compiler will guarantee that accessing global variables, local variables, and fields of data structures will not cause an alignment fault. The only cases in which C code can generate an alignment fault are when a pointer is cast from one type to another or when packed data structures are used. Alignment faults can also be generated from assembly language, but it is the assembly programmer's responsibility to ensure alignment faults are not generated.

Example 165 shows an example that will generate an alignment fault on an Itanium processor. The first read access through *SmallValuePointer* is aligned because *LargeValue* is on a 64-bit boundary. However, the second read access though *SmallValuePointer* will generate an alignment fault because *SmallValuePointer* is not on a 32-bit boundary. The problem is that an 8-bit pointer was cast to a 32-bit pointer. Whenever a cast is made from a pointer to a smaller data type to a pointer to a larger data type, there is a chance that the pointer to the larger data type will be unaligned.

```
UINT64  LargeValue;
UINT32  *SmallValuePointer;
UINT32  SmallValue;

SmallValuePointer  = (UINT32 *)&LargeValue;
SmallValue         = *SmallValuePointer;                    // Works
SmallValuePointer  = (UINT32 *)((UINT8 *)&LargeValue + 1);
SmallValue         = *SmallValuePointer;                    // Faults
```

Example 165. Pointer-Cast Alignment Fault

Example 166 shows the same example as Example 165, but it has been modified to prevent the alignment fault. The second read access through *SmallValuePointer* is replaced with a macro that treats the 32-bit value as an array of bytes. The individual bytes are read and combined into a 32-bit value. The generated object code is larger and slower, but it is functional on both IA-32 and Itanium processors.

```
#define UNPACK_UINT32(a) (UINT32)( (((UINT8 *) a)[0] <<  0) |     \
                                   (((UINT8 *) a)[1] <<  8) |     \
                                   (((UINT8 *) a)[2] << 16) |     \
                                   (((UINT8 *) a)[3] << 24) )

UINT64  LargeValue;
UINT32  *SmallValuePointer;
UINT32  SmallValue;

SmallValuePointer  = (UINT32 *)&LargeValue;
SmallValue         = *SmallValuePointer;                    // Works
SmallValuePointer  = (UINT32 *)((UINT8 *)&LargeValue + 1);
SmallValue         = UNPACK_UINT32 (SmallValuePointer);     // Works
```

Example 166. Corrected Pointer-Cast Alignment Fault

Example 167 shows another example that will generate an alignment fault on an Itanium processor. The first read access from *MyStructure.First* will always work because the 8-bit value is always aligned. However, the second read access from *MyStructure.Second* will always fail because the 32-bit value will never be aligned on a 4-byte boundary.

```
#pragma pack(1)
typedef struct {
  UINT8   First;
  UINT32  Second;
} MY_STRUCTURE;
#pragma pack()

MY_STRUCTURE  MyStructure;
UINT8         FirstValue;
UINT32        SecondValue;

FirstValue  = MyStructure.First;   // Works
SecondValue = MyStructure.Second;  // Faults
```

Example 167. Packed Structure Alignment Fault

Example 168 shows the same example as Example 167, but it has been modified to prevent the alignment fault. The second read access from *MyStructure.Second* is replaced with a macro that treats the 32-bit value as an array of bytes. The individual bytes are read and combined into a 32-bit value. The generated object code is larger and slower, but it is functional on both IA-32 and Itanium processors.

```
#define UNPACK_UINT32(a) (UINT32)( (((UINT8 *) a)[0] <<  0) |     \
                                   (((UINT8 *) a)[1] <<  8) |     \
                                   (((UINT8 *) a)[2] << 16) |     \
                                   (((UINT8 *) a)[3] << 24) )

#pragma pack(1)
typedef struct {
  UINT8   First;
  UINT32  Second;
} MY_STRUCTURE;
#pragma pack()

MY_STRUCTURE  MyStructure;
UINT8         FirstValue;
UINT32        SecondValue;

FirstValue  = MyStructure.First;                    // Works
SecondValue = UNPACK_UINT32(&MyStructure.Second);   // Works
```

Example 168. Corrected Packed Structure Alignment Fault

If a data structure is copied from one location to another, then both the source and the destination pointers for the copy operation should be aligned on a 64-bit boundary for Itanium platforms. The **gBS->CopyMem()** service can handle unaligned copy operations, so an alignment fault will not be generated by the copy operation itself. However, if the fields of the data structure at the destination location are accessed, they may generate alignment faults if the destination address is not aligned on a 64-bit boundary. There are cases where an aligned structure may be copied to an unaligned destination, but the fields of the destination buffer must not be accessed after the copy operation is completed. An example of this case is when a packed structure is being built that will be stored on disk or transmitted on a network.

In some cases, it may be necessary to copy a data structure from an unaligned source location to an aligned destination location so that the fields of the data structure can be accessed without generating an alignment fault. Two examples of this scenario are Parsing EFI device path nodes and Parsing network packets.

The device path nodes in an EFI device path are packed together so they will take up as little space as possible when they are stored in environment variables such as *ConIn*, *ConOut*, *StdErr*, *Boot####*, and *Driver####*. As a result, individual device path nodes may not be aligned on a 64-bit boundary. EFI device paths or device paths nodes can be passed around as opaque data structures, but whenever the fields of a device path node need to be accessed, the device path node must be copied to a location that is guaranteed to be on a 64-bit boundary. Likewise, network packets are packed so they take up as little space as possible, so as each layer of a network packet is examined, it may need to be copied to a 64-bit aligned location before the individual fields of the packet are examined.

Example 169 shows an example of a function that parses an EFI device path and extracts the 32-bit HID and UID from an ACPI device path node. This example will generate an alignment fault if *DevicePath* is not aligned on a 32-bit boundary.

```
VOID
GetAcpiHidUid (
  EFI_DEVICE_PATH_PROTOCOL  *DevicePath,
  UINT32                    *Hid,
  UINT32                    *Uid
  )
{
  ACPI_HID_DEVICE_PATH  *AcpiDevicePath;
  AcpiDevicePath = (ACPI_HID_DEVICE_PATH *)DevicePath;
  *Hid = AcpiDevicePath.HID;            // May fault
  *Uid = AcpiDevicePath.UID;            // May fault
}
```

Example 169. EFI Device Path Node Alignment Fault

Example 170 shows the corrected version of Example 169. Because the alignment of *DevicePath* cannot be guaranteed, the solution is to copy the ACPI device path node from *DevicePath* into an ACPI device path node structure that is declared as the local variable *AcpiDevicePath*. A structure declared as a local variable is guaranteed to be on a 64-bit boundary on Itanium platforms. The fields of the ACPI device path node can then be safely accessed without generating an alignment fault.

```
VOID
GetAcpiHidUid (
  EFI_DEVICE_PATH_PROTOCOL  *DevicePath,
  UINT32                    *Hid,
  UINT32                    *Uid
  )

{
  ACPI_HID_DEVICE_PATH  AcpiDevicePath;

  gBS->CopyMem(&AcpiDevicePath, DevicePath, sizeof
(ACPI_HID_DEVICE_PATH));
  *Hid = AcpiDevicePath.HID;            // Guaranteed to work
  *Uid = AcpiDevicePath.UID;            // Guaranteed to work
}
```

Example 170. Corrected EFI Device Path Node Alignment Fault

21.2 Accessing a 64-Bit BAR in a PCI Configuration Header

Another source of alignment faults is when 64-bit BAR values are accessed in a PCI configuration header. A PCI configuration header has room for up to six 32-bit BAR values or three 64-bit BAR values. A PCI configuration header may also contain a mix of both 32-bit BAR values and 64-bit BAR values. All 32-bit BAR values are guaranteed to be on a 32-bit boundary. However, 64-bit BAR values may be on a 32-bit boundary or a 64-bit boundary. As a result, every time a 64-bit BAR value is accessed, it must be assumed to be on a 32-bit boundary to guarantee that an alignment fault will not be generated. The following are a couple of methods that may be used to prevent an alignment fault when a 64-bit BAR value is extracted from a PCI configuration header:

Method 1:

Use **gBS->CopyMem()** to transfer the BAR contents into a 64-bit aligned location.

Method 2:

Collect the two 32-bit values that compose the 64-bit BAR and combine them into a 64-bit value.

Example 171 below shows the incorrect method of extracting a 64-bit BAR from a PCI configuration header and two correct methods.

```
UINT64
Get64BitBarValue (
  PCI_TYPE00  *PciConfigurationHeader,
  UINTN       BarOffset
  )

{
  UINT64  *BarPointer64;
  UINT32  *BarPointer32;
  UINT64  BarValue;

  BarPointer64 = (UINT64 *)((UINT8 *)PciConfigurationHeader + BarOffset);
  BarPointer32 = (UINT32 *)((UINT8 *)PciConfigurationHeader + BarOffset);

  //
  // Wrong. May cause an alignment fault.
  //
  BarValue = *BarPointer64;

  //
  // Correct. Guaranteed not to generate an alignment fault.
  //
  gBS->CopyMem (&BarValue, BarPointer64, sizeof (UINT64));

  //
  // Correct. Guaranteed not to generate an alignment fault.
  //
  BarValue = (UINT64)(*BarPointer32 | LShiftU64 (*(BarPointer32 + 1),
32));

  return BarValue;
}
```

Example 171. Accessing a 64-Bit BAR in a PCI Configuration Header

21.3 Assignment and Comparison Operators

There are issues if a data value is cast from a larger size to a smaller size. In these cases, the upper bits of the larger values are stripped. In general, this stripping will cause a compiler warning, so these are easy issues to catch. However, there are a few cases where everything compiles free of errors and warnings on IA-32 and generates errors or warnings on the Itanium processor. The only way to guarantee that these errors are caught early is to compile for both IA-32 and Itanium processors during the entire development process. When one of these warnings is generated by an Intel® Itanium® compiler, the warning can be eliminated by explicitly casting the larger data type to the smaller data type. However, the developer needs to make sure that this casting is the right solution, because the upper bits of the larger data value will be stripped.

Example 172 shows several examples that will generate a warning and how to eliminate the warning with an explicit cast. The last example is the most interesting one because it does not generate any warnings on IA-32, but it will on Itanium

architecture. This difference is because a **UINTN** on IA-32 is identical to **UINT32**, but **UINTN** on Itanium architecture is identical to a **UINT64**.

```
UINT8    Value8;
UINT16   Value16;
UINT32   Value32;
UINT64   Value64;
UINTN    ValueN;

Value8  = Value16;           // Warning generated on IA-32 and Itanium
Value8  = (UINT8)Value16;    // Works, upper 8 bits stripped
Value16 = Value8;            // Works

Value8  = Value32;           // Warning generated on IA-32 and Itanium
Value8  = (UINT8)Value32;    // Works, upper 24 bits stripped
Value32 = Value8;            // Works

Value8  = Value64;           // Warning generated on IA-32 and Itanium
Value8  = (UINT8)Value64;    // Works, upper 56 bits stripped
Value64 = Value8;            // Works

Value8  = ValueN;            // Warning generated on IA-32 and Itanium
Value8  = (UINT8)ValueN;     // Works, upper 24 bits stripped on IA-32.
                             // Upper 56 bits stripped on Itanium
ValueN  = Value8;            // Works

Value32 = ValueN;            // Works on IA-32, warning generated on
Itanium
Value32 = (UINT32)ValueN;    // Works on IA-32, upper 32 bits stripped on
                             // Itanium
```

Example 172. Assignment Operation Warnings

Example 173 is very similar to Example 172, except the assignments have been replaced with comparison operations. The same issues shown will be generated by all the comparison operators, including **>**, **<**, **>=**, **<=**, **!=**, and **==**. The solution is to cast one of the two operands to be the same as the other operand. The first four cases are the ones that work on IA-32 with no errors or warnings but generate warnings on Itanium architecture. The next four cases resolve the issue by casting the first operand, and the last four cases resolve the issue by casting the second operand. Care must be taken when casting the correct operand, because a cast from a larger data type to a smaller data type will strip the upper bits of the operand. When a cast is performed to **INTN** or **UINTN**, a different number of bits will be stripped for IA-32 and Itanium architecture.

```
UINT64  ValueU64;
UINTN   ValueUN;
INT64   Value64;
INTN    ValueN;

if (ValueU64 == ValueN)   {}  // Works on IA-32, warning generated on
Itanium
if (ValueUN  == Value64)  {}  // Works on IA-32, warning generated on
Itanium
if (Value64  == ValueUN)  {}  // Works on IA-32, warning generated on
Itanium
if (ValueN   == ValueU64) {}  // Works on IA-32, warning generated on
Itanium

if ((INTN)ValueU64 == ValueN)   {}  // Works on both IA-32 and Itanium
if ((INT64)ValueUN == Value64)  {}  // Works on both IA-32 and Itanium
if ((UINTN)Value64 == ValueUN)  {}  // Works on both IA-32 and Itanium
if ((UINT64)ValueN == ValueU64) {}  // Works on both IA-32 and Itanium

if (ValueU64 == (UINT64)ValueN) {}  // Works on both IA-32 and Itanium
if (ValueUN  == (UINTN)Value64) {}  // Works on both IA-32 and Itanium
if (Value64  == (INT64)ValueUN) {}  // Works on both IA-32 and Itanium
if (ValueN   == (INTN)ValueU64) {}  // Works on both IA-32 and Itanium
```

Example 173. Comparison Operation Warnings

21.4 Casting Pointers

Pointers can be cast from one pointer type to another pointer type. However, pointers should never be cast to a fixed-size data type, and fixed-size data types should never be cast to pointers. The size of a pointer varies between IA-32 and Itanium processors. If any assumptions are made that a pointer to a function or a pointer to a data structure is a 32-bit value, then that code will not run on Itanium-based platforms with physical memory above 4 GB. These issues difficult are to catch, because explicit casts are required to cast a fixed-width type to a pointer or vice versa. Once these explicit type casts are introduced, no compiler warnings or errors will be generated. In fact, the code may execute just fine on IA-32 platforms and Itanium-based platforms with physical memory below 4 GB. The only failing case will be when the code is tested on an Itanium-based system with physical memory above 4 GB. The symptom is typically a processor exception that results in a system hang or reset. Example 174 below shows some good and bad examples of casting pointers. The first group is casting pointers to pointers. The second group is casting pointers to fixed width types, and the last group is casting fixed width types to pointers. There is one exception to this rule that applies to IA-32 and Itanium processors. The data types **INTN** and **UINTN** are the exact same size of pointers on both IA-32 and Itanium-based platforms, which means that a pointer can be cast to or from **INTN** or **UINTN** without any adverse side effects. However, ANSI C does not require function pointers to be the same size as data pointers, and function pointers and data pointers are not required to be the same size as **INTN** or **UINTN**. As a result, this exception does not apply to all processors.

```
typedef struct {
  UINT8   First;
  UINT32  Second;
} MY_STRUCTURE;

MY_STRUCTURE  *MyStructure;
UINT8         ValueU8;
UINT16        ValueU16;
UINT32        ValueU32;
UINT64        ValueU64;
UINTN         ValueUN;
INT64         Value64;
INTN          ValueN;
VOID          *Pointer;

//
// Casting pointers to pointers
//
Pointer     = (VOID *)MyStructure;      // Good.
MyStructure = (MY_STRUCTURE *)Pointer;  // Good.

//
// Casting pointers to fixed width types
//
ValueU8  = (UINT8)MyStructure;     // Bad. Strips upper 24 bits on IA-32
and
                                   // upper 56 bits on Itanium.
ValueU16 = (UINT16)MyStructure;    // Bad. Strips upper 16 bits on IA-32
and
                                   // upper 48 bits on Itanium.
ValueU32 = (UINT32)MyStructure;    // Bad. Works on IA-32, but strips
upper
                                   // 32 bits on Itanium.
ValueU64 = (UINT64)MyStructure;    // OK. Works on IA-32 and Itanium
Value64  = (INT64)MyStructure;     // OK. Works on IA-32 and Itanium
ValueUN  = (UINTN)MyStructure;     // Good. Works on IA-32 and Itanium
ValueN   = (INTN)MyStructure;      // Good. Works on IA-32 and Itanium

//
// Casting fixed width types to pointers
//
MyStructure = (MY_STRUCTURE *)ValueU8;   // Bad
MyStructure = (MY_STRUCTURE *)ValueU16;  // Bad
MyStructure = (MY_STRUCTURE *)ValueU32;  // Bad. Works on IA-32, only
works on
                                         // Itanium-based platforms with
< 4 GB
MyStructure = (MY_STRUCTURE *)ValueU64;  // OK. Works on IA-32 and
Itanium
MyStructure = (MY_STRUCTURE *)Value64;   // OK. Works on IA-32 and
Itanium
MyStructure = (MY_STRUCTURE *)ValueUN;   // Good. Works on IA-32 and
Itanium
MyStructure = (MY_STRUCTURE *)ValueN;    // Good. Works on IA-32 and
Itanium
```

Example 174. Casting Pointer Examples

21.5 EFI Data Type Sizes

There are a few EFI data types that are different sizes on IA-32 and Itanium architecture, as follows:

- Pointers

- Enumerations
- **INTN**
- **UINTN**

These differing types also mean that any complex types, such as unions and data structures, that are composed of these base types will also have different sizes on IA-32 and Itanium architecture. These differences must be understood whenever the **sizeof()** operator is used. If a union or data structure is required that does not change size between IA-32 and Itanium architecture, see o for a summary of the EFI data types that are available to all UEFI applications and UEFI drivers.

21.6 Negative Numbers

Negative numbers are not the same on IA-32 and Itanium processors. Negative numbers are type **INTN**, and **INTN** is a 4-byte container on IA-32 and an 8-byte container on the Itanium processor. For example, **-1** on IA-32 is **0xFFFFFFFF**, and **-1** on the Itanium processor is **0xFFFFFFFFFFFFFFFF**. Care must be taken when assigning or comparing negative numbers. Example 175 shows an example that compiles without errors or warnings on both IA-32 and Itanium processors but behaves very differently on IA-32 than it does on the Itanium processor.

```
UINT32  ValueU32;

ValueU32 = 0xFFFFFFFF;

if ((INTN)ValueU32 == -1) {
  Print(L"Equal\n");       // This message is printed on IA-32 and not
Itanium
} else {
  Print(L"Not Equal\n");   // This message is printed on Itanium and not
IA-32
}
```

Example 175. Negative Number Example

21.7 Returning Pointers in a Function Parameter

Example 176 shows a bad example for casting pointers. The function **MyFunction()** simply returns a 64-bit value in an **OUT** parameter that is assigned from a 32-bit input parameter. There is nothing wrong with **MyFunction()**. The problem is when **MyFunction()** is called. Here, the address of **B**, a 32-bit container, is cast to a pointer to a 64-bit container and passed to **MyFunction()**. **MyFunction()** writes to 64 bits starting at **B**. This location happens to overwrite the value of **B** and the value of **A** in the calling function. The first **Print()** correctly shows the values of **A** and **B**. The second **Print()** shows that **B** was given **A**'s original value, but the contents of **A** were destroyed and overwritten with a 0. The cast from **&B** to a **(UINT64 *)** is the problem here. This code compiles without errors or warnings in both IA-32 and Itanium processors. It executes on IA-32 with these unexpected side effects. It might run on Itanium processors, but it depends on if **&B** is 64-bit aligned or not. There is a 50 percent chance that it will generate an alignment fault on an Itanium processor. If it does not generate an alignment fault, then it will get the same unexpected results that occurred on IA-32. This porting issue is not specific to the Itanium processor. However,

because 64-bit quantities are more likely to be used in UEFI drivers, this issue is an important one to consider when doing EFI development work.

```
EFI_STATUS
MyFunction (
  IN  UINT32  ValueU32,
  OUT UINT64  *ValueU64
  )

{
  *ValueU64 = (UINT64)ValueU32;
  return EFI_SUCCESS;
}


UINT32  A;
UINT32  B;

A = 0x11112222;
B = 0x33334444;
Print(L"A = %08x  B = %08x\n",A,B);  // Prints "A = 11112222  B =
33334444"
MyFunction (A, (UINT64 *)(&B));
Print(L"A = %08x  B = %08x\n",A,B);  // Prints "A = 00000000  B =
11112222"
```

Example 176. Casting OUT Function Parameters

21.8 Array Subscripts

In general, array subscripts should be of type **INTN** or **UINTN**. Using these types will avoid problems if an array subscript is decremented below **0**. If a **UINT32** is used as an array subscript and is decremented below 0, it is decremented to **0xFFFFFFFF** on IA-32 and **0x00000000FFFFFFFF** on the Itanium processor. These subscript values are very different. On IA-32, this value is the same indexing element as **-1** of the array. However, on the Itanium processor, this value is the same indexing element as **0xFFFFFFFF** of the array. If an **INTN** or **UINTN** is used instead of a **UINT32** for the array subscript, then this problem goes away. When a **UINTN** is decremented below **0**, it is decremented to **0xFFFFFFFF** on IA-32 and **0xFFFFFFFFFFFFFFFF** on the Itanium processor. These values are both the same indexing element as **-1** of the array. Example 177 below show two examples of array subscripts. The first one works on IA-32 but faults on Itanium processors. The second example is rewritten to work properly on both IA-32 and Itanium processors.

```
UINT32  Index;
CHAR8   Array[] = "ABCDEFGHIJKLIMNOPQRSTUVWXYZ";
CHAR8   *MyArray;

MyArray = &(Array[5]);
Index = 0;
Print(L"Character = %c\n",Array[Index-1]);  // Works on IA-32, faults on
Itanium


UINTN   Index;
CHAR8   Array[] = "ABCDEFGHIJKLIMNOPQRSTUVWXYZ";
CHAR8   *MyArray;

MyArray = &(Array[5]);
Index = 0;
Print(L"Character = %c\n",Array[Index-1]);  // Works on IA-32 and Itanium
```

Example 177. Array Subscripts Example

21.9 Piecemeal Structure Allocations

Structures should always be allocated using the **sizeof()** operator on the name of the structure. The sum of the sizes of the structure's members should never be used because it does not take into account the padding that the compiler introduces to guarantee alignment. Example 178 shows two examples for allocating memory for a structure. The first one is incorrect and the second allocation is correct.

```
typedef struct {
  UINT8   Value8;
  UINT64  Value64;
} MY_STRUCTURE;

MY_STRUCTURE  *MyStructure;

//
// Wrong. This will only allocate 9 bytes, but MyStructure is 16 bytes
//
Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (UINT8) + sizeof (UINT64),
                (VOID **)&MyStructure
                );

//
// Correct. This will allocate 16 bytes for MyStructure.
//
Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (MY_STRUCTURE),
                (VOID **)&MyStructure
                );
```

Example 178. Piecemeal Structure Allocation

21.10 Speculation and Floating Point Register Usage

Itanium processors support speculative memory accesses and a large number of floating point registers. UEFI drivers that are compiled for Itanium processors must follow the calling conventions defined in the *SAL Specification.* This specification allows

only the first 32 floating point registers to be used and defines the amount of speculation support that a platform is required to implement for the preboot environment. These requirements mean that the correct compiler and linker switches must be used to guarantee that these calling conventions are followed. The *EDK* includes the correct compiler and linker settings for several tool chains for the Itanium processor. These settings may have to be adjusted if a newer or different tool chain is used. Table 34 shows the compiler flags for a few different compilers.

Table 34. Compiler Flags

Compiler	Optimizations	Description
Intel Itanium Compiler 5.01	Off	/X /Zl /Zi /Od /W3 /QIPF_fr32
	On	/X /Zl /O1 /W3 /QIPF_fr32
Intel® C++ Compiler 7.1 for Windows*	Off	/X /Zi /Zl /Od /W3 /WX /QIPF_fr32
	On	/X /Zl /O1 /W3 /WX /QIPF_fr32

21.11 Memory Ordering

The store model of the processor or the store model for a bus master in an I/O subsystem may be weakly ordered. Weak ordering of processor store cycles has been the source of several difficult bugs in a number of drivers. It is easy to imagine that this issue could be fairly widespread in drivers written for Itanium processors at both the OS and EFI level. See *A Formal Specification of Intel Itanium Processor Family Memory Ordering* for a detailed discussion of this topic, which is also discussed in the *Intel® Itanium® Architecture Software Developer Manuals, volumes 1–4.*

The classic case where strong ordering versus weak ordering produces different results is when there is a memory-based FIFO and a shared bus master "doorbell" register that is shared by all additions to the FIFO. In this common implementation, the driver (producer) formats a new request descriptor and, as its last logical operation, writes the value indicating the entry is valid.

This mechanism becomes a problem if a new request is being added to the FIFO while the bus master is checking the next FIFO entry's valid flag. It is possible for the "last write" issued by the processor (that turns on the valid flag) to be posted to memory before the logically earlier writes that finish initializing the FIFO/request descriptor.

The solution in this case is to ensure that all pending memory writes have been completed before the "valid flag" is enabled. There are two techniques to avoid this problem:

Technique 1:

Declare the whole structures with the C language "volatile" attribute. The compiler will ensure that strong ordering is used for all operations in this case.

Technique 2:

Use the **MEMORY_FENCE()** macro before setting the valid flag. This macro calls the "**__mf();**" intrinsic, which will ensure that all previous stores are posted. The intrinsic call requires that a **#pragma intrinsic (__mfa)** statement be defined. The EDK sets up this macro in the **\Foundation\Include\<<Architecture>>\efibind.h** header file, where <<architecture>> is one of EBC, IA-32, IPF, or x64.

The second solution is typically preferred for readability because the intent is clearer. A volatile declaration tends to hide what was needed, because it is not part of the affected code (because it is off in a structure definition). In addition, using the volatile declaration could impact the driver's performance because all memory transactions to the structure would be strongly ordered (ordered memory transactions are slower).

Note: *If a driver is executing in the EBC environment, the EBC interpreter ensures that all memory transactions are strongly ordered. Technically, EBC drivers do not need to use the **MEMORY_FENCE()** macro. However, for portability to non-EBC environments and for readability, the use of the **MEMORY_FENCE()** macro is strongly encouraged.*

Many driver/bus master pairs do not exhibit this issue, based on the style of their driver/bus master interactions.

The easy/safe case is when the driver builds a structure and the bus master will not access that structure until the (exclusive to this request, or only one request outstanding) bus master "doorbell" is rung, and the doorbell resides on a PCI device. This mechanism works because **PciIo->IoWrite()** and **PciIo->MemWrite()** are also memory fence operations.

Another "safe" variation is a memory FIFO of host requests, and the host writes the current FIFO producer index to the bus master's doorbell. This mechanism is safe because the bus master will not attempt to read the FIFO until the corresponding index has been written, and the bus master write is the memory fence.

Note: *This mechanism is really a variation of the "easy/safe case" above (two paragraphs before this note), because the producer index makes the doorbell write exclusive to a single request; i.e., the bus master does not "read ahead."*

Another variation in the "unique doorbell per request" category is the bus master doorbell that is really a bus-master-based FIFO. The driver typically writes the address of the request to this FIFO register. This method is commonly used in the "I_2O" model but is used by several vendors without being I_2O compliant.

21.12 Helpful Tools

To catch possible issues with assigning or comparing values of different sizes, UEFI drivers should always be compiled with fairly high warning levels. For example, the Microsoft Visual Studio tool chain supports the /WX and /W3 or /W4 compiler flags. The /WX flag will cause any compile time warnings to generate an error, so the build will stop when a warning is generated. The /W3 and /W4 flags set the warning level to 3 and 4 respectively. At these warning levels, any size mismatches in assignments and comparisons will generate a warning. With the /WX flag, the compile will stop when these size mismatches are detected.

If an UEFI driver is being developed for an IA-32 system and is planned to be ported to the Itanium processor, then it is always a good idea to compile the UEFI driver with an Itanium compiler during the development process to make sure the code is clean when validation on the Itanium processor is begun. By using the /WX and /W3 or /W4 compiler flags, any size mismatches that are generated by only 64-bit code will be detected.

22 EFI Byte Code Porting Considerations

There are a few considerations to keep in mind when writing drivers that may be ported to EBC. This chapter describes these considerations in detail and, where applicable, provides solutions to address them. If UEFI drivers are implemented with these considerations in mind, then porting a native driver to EBC may simply require a recompile using the Intel® C Compiler for EFI Byte Code.

22.1 No Assembly Support

The only tools that are available for EBC are a C compiler and a PE/COFF linker. There are no EBC assemblers available, and there are no plans to produce an EBC assembler. This lack of an EBC assembler is actually by design, because the EBC instruction set was optimized with a C compiler in mind. If the driver is being ported to EBC, all assembly language for IA-32 or Itanium processors must be converted to C.

22.2 No Floating Point Support

There is no floating-point support in the EBC Virtual Machine, which means that the type **`float`** is not supported in the Intel C Compiler for EFI Byte Code. If an UEFI driver is being ported to EBC and requires floating-point math, then the driver must be converted to fixed-point math using integer operands and operators.

22.3 No C++ Support

The Intel C Compiler for EFI Byte Code does not support C++. If there is any C++ code in an UEFI driver being ported to EBC, then that C++ code must be converted to C.

22.4 EFI Data Type Sizes

22.4.1 Sizeof()

In some cases, **`sizeof()`** is computed at runtime for EBC code, whereas **`sizeof()`** is never computed at runtime for native code. Because pointers, the EFI data types **`INTN`** and **`UINTN`**, the C type **`long`**, and EFI status codes are different sizes on different instruction set architectures, an EBC image must adapt to the platform on which it is

executing. Example 179 below shows several examples of simple and complex data types.

```
typedef enum {Red, Green, Blue} COLOR_TYPE;

typedef struct {
  UINT64  ValueU64;
  UINT32  ValueU32;
  UINT16  ValueU16;
  UINT8   ValueU8;
} FIXED_STRUCTURE;

typedef struct {
  UINTN   ValueUN;
  VOID    *Pointer;
  UINT64  ValueU64;
  UINT32  ValueU32;
} VARIABLE_STRUCTURE;

Size = sizeof (UINT64);              //  8 bytes on IA-32,  8 bytes on
Itanium
Size = sizeof (UINT32);              //  4 bytes on IA-32,  4 bytes on
Itanium
Size = sizeof (UINT16);              //  2 bytes on IA-32,  2 bytes on
Itanium
Size = sizeof (UINT8);               //  1 byte  on IA-32,  1 byte  on
Itanium
Size = sizeof (UINTN);               //  4 bytes on IA-32,  8 bytes on
Itanium
Size = sizeof (INTN);                //  4 bytes on IA-32,  8 bytes on
Itanium
Size = sizeof (COLOR_TYPE);          //  4 bytes on IA-32,  8 bytes on
Itanium
Size = sizeof (VOD *);               //  4 bytes on IA-32,  8 bytes on
Itanium
Size = sizeof (FIXED_STRUCTURE);     // 15 bytes on IA-32, 15 bytes on
Itanium
Size = sizeof (VARIABLE_STRUCTURE); // 20 bytes on IA-32, 28 bytes on
Itanium
```

Example 179. Size of EBC Data Types

For the types that return different sizes for IA-32 and Itanium processors, the EBC compiler generates code that computes the correct values at runtime.

22.4.2 Initialization using processor-dependent value

In a native compile the value of **`sizeof(UINTN)`** will be calculated by the compiler at compile time. This can be done because the compiler already knows the instruction set architecture. The EBC compiler cannot do that in the same way. It will instead generate some code to calculate this value at execution time.

This limitation means that EBC code cannot use **`sizeof(UINTN)`**, **`sizeof(INTN)`**, and **`sizeof(VOID*)`** in constant expressions.

Note: *The EFI error status codes are also defined with different length on different instruction set architectures This means that EFI error status codes cannot be used in constant expression.*

The code from Example 180 will fail in EBC.

```
//
// Global variable definition
//
UINTN IntegerSize = sizeof(UINTN);              // Error
UINTN PointerSize = sizeof(VOID*);              // Error

EFI_STATUS Status = EFI_INVALID_PARAMETER;      // Error

VOID Function (VOID)
{
  EFI_STATUS FuncStatus;

  FuncStatus = SubFunc ();
  swtich (FuncStatus) {
    case EFI_INVALID_PARAMETER:                 // Error
      break;
    default:
      break;
  }
}
```

Example 180 Code that will fail in EBC

22.4.2.1 CASE Statements

Because pointers and the data types **INTN** and **UINTN** are different sizes on different instruction set architectures and case statements are determined at compile time; the **sizeof()** function cannot be used in a **case** statement with an indeterminately sized data type because the **sizeof()** function cannot be evaluated to a constant by the EBC compiler at compile time. EFI status codes such as **EFI_SUCCESS** and **EFI_UNSUPPORTED** are defined differently on different instruction set architectures. As a result, EFI status codes cannot be used in **case** expressions. Example 181 shows examples of case statements.

```
UINTN Value;

switch (Value) {
case 0:                 // Works
  break;
case sizeof (UINT16):   // Works because sizeof (UINT16) is always 2
  break;
case sizeof (UINTN):    // Compiler error - sizeof (UINTN) is not constant
  break;
case EFI_UNSUPPORTED:   // Compiler error - EFI_UNSUPPORTED is not
constant
  break;
}
```

Example 181. Case Statements

22.5 Stronger Type Checking

The EBC compiler performs stronger type checking than some other IA-32 and Itanium compilers. As a result, code that compiles without any errors or warnings on an IA-32 or Itanium compiler may generate warnings with the EBC compiler. Example 182 shows two common examples, using **gBS->AllocatePool()** and **gBS->OpenProtocol()**, from UEFI drivers that will generate warnings with the EBC compiler and how these examples can be fixed.

```
typedef struct {
  UINT8   Value8;
  UINT64  Value64;
} MY_STRUCTURE;

EFI_STATUS                    Status;
EFI_DRIVER_BINDING_PROTOCOL   *This;
EFI_HANDLE                    ControllerHandle;
EFI_GUID                      gEfiBlockIoProtocolGuid;
EFI_BLOCK_IO_PROTOCOL         *BlockIo;
MY_STRUCTURE                  *MyStructure;

Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiBlockIoProtocolGuid,
                &BlockIo,                       // Compiler warning
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );

Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiBlockIoProtocolGuid,
                (VOID **)&BlockIo,              // No compiler warning
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );

Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (MY_STRUCTURE),
                &MyStructure                    // Compiler warning
                );

Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (MY_STRUCTURE),
                (VOID **)&MyStructure           // No compiler warning
                );
```

Example 182. Stronger Type Checking

22.6 UEFI driver Entry Point

The entry point to every EBC driver is a function called **EfiStart()**. The **EfiStart()** function performs the runtime initialization of the UEFI driver and then calls **EfiMain()**. The entry point that is declared in the **Make.inf** file is always renamed to **EfiMain()**. These details are hidden from the developer by the build environment. The symbols **EfiStart()** and **EfiMain()** are used by the EBC build environment. As a result, these two reserved symbols cannot be used as function names or variable names in an UEFI driver implementation.

22.7 Memory Ordering

The EBC interpreter ensures that all memory transactions are strongly ordered. EBC drivers are not required to use the **MEMORY_FENCE()** macro when strong ordering is

required. However, to guarantee that an UEFI driver is portable to non-EBC execution environments, the use of the **MEMORY_FENCE()** macro is strongly encouraged.

22.8 Performance Considerations

All EBC executables require an EBC Virtual Machine interpreter to be executed. Because all EBC executables are running through an interpreter, they will run slower than native EFI executables. As a result, an UEFI driver that is compiled with an EBC compiler should be optimized for performance to improve the usability of the UEFI driver. Chapter 20 covers speed optimization techniques that can be used to improve the performance of all UEFI drivers.

The simplest way to maximize the speed of a UEFI EBC driver is to use as many of the native helper functions as possible. When the native function is executing it will be executing natively and will run much faster than the same code executing through the EBC interpreter.

22.8.1 Performance Considerations for data types

Do not statically initialize any variables that are native length (pointers, UINTN, INTN). They will get initialized at runtime and that slows everything else down. If you have to have them in a table it may be better to make them always 64 bits long.

This can have a significant effect on such driver types as graphics drivers that have large tables to initialize at the beginning.

This can be measured using the Microsoft linker tool. Use the "link –dump <filename>" command to dump the different parts of your .EFI file. The goal is to minimize the variable init section while maximizing the data section of the file.

22.9 Comparing natural and normal integers

A Natural Integer is defined as the native length for the runtime instruction set architecture. That is this integer is a 32-bit value on a 32-bit platform, and is a 64-bit value on a 64-bit platform. It is equal to **UINTN**/**INTN**/**VOID*** type in UEFI specification.

A Normal Integer is an integer of predetermined length. An **int** integer is a 32-bit value on all platforms and a **__int64** is a 64-bit value all platforms.

Because of the limitation of architecture, arithmetic calculation between natural integers and normal integers should be avoided. It is described in the Known Limitations section in *Release Notes for the EBC compiler*.

The code in Example 182 will ASSERT if compiled in EBC depending on the runtime instruction set architecture.

```
#define MACRO 0x10000000UL        // Value is not important

VOID Function (VOID)
{
  DWORD dw = MACRO;         // dw is 32 bits

  If (dw != MACRO) {
ASSERT();
  }
}
```

Example 183. Macro with data type

There are some possible solutions to this problem with 2 examples below…

- Cast MACRO as a DWORD (**`DWORD dw = (DWORD)MACRO;`**)
- Remove the 'L' qualifier from the #define. (**`#define MACRO 0x10000000U`**);

Neither one of those solutions is very helpful if the code will be compiled by a different compiler since they are both limiting the compile to certain instruction set architectures.

23 Building UEFI drivers

This section describes how to write, compile, and package UEFI drivers in the *EDK* environment.

23.1 UEFI Driver Support Files and Directories

New UEFI drivers are added to the *EDK* source tree in the **\Sample\Bus\<<BUS>>\<<DriverName>>** directory. It is recommended that all UEFI drivers be placed below these directories, but it is not strictly required. Additional subdirectories can be created under this directory to help organize the UEFI drivers into groups.

To add a new UEFI driver to the build environment, do the following:

- Create a new subdirectory.
- Place a **Make.inf** file along with the **.c** and **.h** files in that subdirectory.

No assembly language files are allowed in this directory. If assembly language files are required, they should be placed in instruction set architecture-specific subdirectories.

Example 184 shows (some of) the files that are present in an UEFI driver that consumes the Block I/O Protocol and produces the Disk I/O Protocol. These files are placed in a driver directory called **\Sample\Universal\Disk\DiskIo\Dxe**.

```
EDK\
   Sample\
      Universal\
         Disk\
           DiskIo\
              Dxe\
                 ComponentName.c
                 Diskio.c
                 Diskio.h
                 Make.inf
```

Example 184. Disk I/O Driver Files

The example in Example 184 does not contain any instruction set architecture-specific files. This absence means that this driver is designed to be portable between IA-32, Itanium architecture, x64, and EBC. If an UEFI driver requires instruction set architecture-specific components, then those components should be added in subdirectories below the UEFI driver's main directory. A Separate subdirectory is required for each instruction set architecture that the driver will be compiled for. The Table below lists the directory names that are reserved for the instruction set architecture-specific files.

Table 35. Reserved Directory Names

Directory Name	Notes
IA-32	May contain **.c**, **.h**, and **.asm** files.
IPF	May contain **.c**, **.h**, and **.s** files.
EBC	May contain **.c** and **.h** files.
X64	May contain **.c**, **.h**, and **.asm** files.

Example 185 shows a different UEFI driver from Example 184, and this driver includes instruction set architecture-specific files for three of the supported instruction sets. The files **EbcLowLevel.asm** and **EbcLowLevel.s** contain optimized assembly code to improve the performance of the disk I/O driver. Doing so makes the driver work on all three supported architectures, but the UEFI driver takes longer to develop and is more difficult to maintain if any changes are required in the instruction set architecture-specific components. If possible, an UEFI driver should be implemented in C with no instruction set architecture-specific files, which will reduce the development time, reduce maintenance costs, and increase portability.

```
Sample\
    Universal\
        Ebc\
            Dxe\
                EbcInt.c
                EbcInt.h
                EbcExecute.c
                EbcExecure.h
                Ebc.inf
                IA-32\
                    EbcSupport.c
                    EbcLowLevel.asm
                    IA-32Math.asm
                Ipf\
                    EbcSupport.c
                    IfpMul.s
                    IpfMath.c
                    EbcLowLevel.s
                x64\
                    x64Math.c
                    EbcSupport.c
                    EbcLowLevel.asm
```

Example 185. EBC Driver with Instruction set architecture-Specific Files

23.1.1 Make.inf File

The **Make.inf** file specifies the following:

- A list of source files
- The path to the include directories
- The path to the library directories
- The entry point for the UEFI driver
- The name of the executable UEFI driver image

relative path names with "`..`" are not allowed in any of the sections. When relative path names are used, it makes it very difficult to move a component to a different location in the source tree. Likewise, `#include` statements in `.c` and `.h` files should not use relative path names with "`..`" for the same reason.

Note: *Each of the sections of the .INF file can have a .common, .IA-32, .x64, and .ipf. The example for the sources will explain the different section types, but they can be done with each section type, not just the sources.*

Example 186 shows the contents of the `Make.inf` file for the driver from Example 184. Example 187 shows the `Make.inf` file for the driver from Example 185. The full source code for Disk I/O Driver is included in Appendix D for reference.

```
#/*++
#
# Copyright (c) 2004, Intel Corporation
# All rights reserved. This program and the accompanying materials
# are licensed and made available under the terms and conditions of the
BSD License
# which accompanies this distribution.  The full text of the license may
be found at
# http://opensource.org/licenses/bsd-license.php
#
# THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
# WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.
#
#  Module Name:
#
#    DiskIo.inf
#
#  Abstract:
#
#    Component description file for DiskIo module.
#
--*/

[defines]
BASE_NAME            = DiskIo
FILE_GUID            = CA261A26-7718-4b9b-8A07-5178B1AE3A02
COMPONENT_TYPE       = BS_DRIVER

[sources.common]
  diskio.c
  diskio.h
  ComponentName.c

[libraries.common]
  EfiProtocolLib
  EfiDriverLib

[includes.common]
  $(EDK_SOURCE)\Foundation\Efi
  $(EDK_SOURCE)\Foundation
  $(EDK_SOURCE)\Foundation\Framework
  .
  $(EDK_SOURCE)\Foundation\Include
  $(EDK_SOURCE)\Foundation\Efi\Include
  $(EDK_SOURCE)\Foundation\Framework\Include
  $(EDK_SOURCE)\Foundation\Include\IndustryStandard
  $(EDK_SOURCE)\Foundation\Core\Dxe
  $(EDK_SOURCE)\Foundation\Library\Dxe\Include

[nmake.common]
  IMAGE_ENTRY_POINT=DiskIoDriverEntryPoint
```

Example 186 Disk I/O Driver Make.inf File

```
#/*++
#
# Copyright (c)  2004 - 2005, Intel Corporation
# All rights reserved. This program and the accompanying materials
# are licensed and made available under the terms and conditions of the
BSD License
# which accompanies this distribution.  The full text of the license may
be found at
# http://opensource.org/licenses/bsd-license.php
#
# THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
# WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.
#
#  Module Name:
#
#    Ebc.inf
#
#  Abstract:
#
#    Component description file for EBC interpreter.
#
--*/

[defines]
BASE_NAME            = Ebc
FILE_GUID            = 13AC6DD0-73D0-11D4-B06B-00AA00BD6DE7
COMPONENT_TYPE       = BS_DRIVER

[sources.IA-32]
  IA-32\EbcLowLevel.asm
  IA-32\IA-32Math.asm
  IA-32\EbcSupport.c

[sources.ipf]
  Ipf\EbcLowLevel.s
  Ipf\IpfMath.c
  Ipf\IpfMul.s
  Ipf\EbcSupport.c

[sources.x64]
  x64\EbcLowLevel.asm
  x64\x64Math.c
  x64\EbcSupport.c

[sources.common]
  EbcInt.c
  EbcInt.h
  EbcExecute.c
  EbcExecute.h

[includes.common]
  $(EDK_SOURCE)\Foundation\Efi
  $(EDK_SOURCE)\Foundation
  $(EDK_SOURCE)\Foundation\Framework
  .
  $(EDK_SOURCE)\Foundation\Core\Dxe
  $(EDK_SOURCE)\Foundation\Include
  $(EDK_SOURCE)\Foundation\Efi\Include
  $(EDK_SOURCE)\Foundation\Framework\Include
  $(EDK_SOURCE)\Foundation\Include\IndustryStandard
  $(EDK_SOURCE)\Foundation\Library\Dxe\Include

[libraries.common]
  EfiProtocolLib
  EfiDriverLib

[nmake.common]
```

```
IMAGE_ENTRY_POINT=InitializeEbcDriver
DPX_SOURCE=Ebc.dxs
```

Example 187. EBC Driver Make.inf

The following sections describe the different sections that are available in the **Make.inf** file.

23.1.1.1 [sources] Sections

Four sources sections are available in a **make.inf** file. Table 36 describes each of these four types of sources sections.

Table 36. Sources Sections Available in a make.inf File

Type of Section	Optional or Required?	Description
[sources.common]	Required	Contains the names of all required **.c** and **.h** files. If possible, a driver should be designed so that this section is the only sources section. If any of the other sources sections are used, then additional work will be required to port the UEFI driver to the other supported instruction set architecture types.
[sources.IA-32]	Optional	Contains the names of the IA-32 specific **.c**, **.h**, and **.asm** files that are in the **IA-32** subdirectory below the UEFI driver's directory.
[sources.epf]	Optional	Contains the names of the Itanium architecture–specific **.c**, **.h**, and **.s** files that are in the **Ipf** subdirectory below the UEFI driver's directory.
[sources.ebc]	Optional	Contains the names of the EFI Byte Code–specific **.c** and **.h** files that are in the **EBC** subdirectory below the UEFI driver's directory.
[sources.x64]	Optional	Contains the names of the x64–specific **.c** and **.h** files that are in the **x64** subdirectory below the UEFI driver's directory.

The example in Example 184 does not contain any instruction set architecture-specific sources, so it contains only a single **[sources]** section. The **Make.inf** file from Example 185 would contain all four sources sections.

23.1.1.2 [includes] Section

The **[includes.common]** section in a **Make.inf** file contains the list of include directories that are required by the UEFI driver. All UEFI drivers require the include directory named "**.**" to include the **.h** files that are listed from the UEFI driver's directory and the instruction set architecture-specific subdirectories. All UEFI drivers also require the **$(EDK_SOURCE)\Foundation\Include** directory to include all the function prototypes and data structure definitions for EFI Boot Services and EFI Runtime Services. If an UEFI driver uses the UEFI driver Library, then the include directory **$(EDK_SOURCE)\Foundation\Library\Dxe\Include** is also required. If the UEFI driver includes any protocols or GUIDs (see o), then the

$(EFI_SOURCE)\Foundation directory is also required. Additional include paths can be added as required. Example 187 uses all four of these include directories.

23.1.1.3 [libraries] Section

The **[libraries.common]** section in a **Make.inf** file contains the list of libraries against which the UEFI driver will be linked. Table 37 lists the libraries that are required when the UEFI driver has different characteristics. If any new libraries are added to the build environment, then they can be added here too.

Table 37. Required Libraries

If the UEFI driver…	Required Library
Uses the UEFI driver Library	**EfiDriverLib**
Includes any protocols (see o)	**EfiProtocolLib**
Includes any GUIDs (see o)	**EfiGuidLib**
Contains any **DEBUG()** macros	**EfiPrintLib**
Requires the use of print functions	**EfiPrintLib**
Requires any string functions from the UEFI driver Library (see o)	**$(EFI_SOURCE)\EDK\Lib\String**

23.1.1.4 [nmake] Section

The **[nmake.common]** section in a **Make.inf** file contains one required line, which is **IMAGE_ENTRY_POINT=**. Any other statements in the **[nmake]** section are added to the **makefile** that is used to build the UEFI driver. This feature can be used to override the default compiler and linker flags.

23.1.1.5 [defines] Section

There are 3 required lines in the **[defines]** section. They are for defining non-code parts of the compiled file. The description of each is in Table 38.

Table 38. [defines] section lines

Line	Value
BASE_NAME	A single-word driver name (no spaces)
FILE_GUID	The GUID for the driver
COMPONENT_TYPE	Either **BS_DRIVER** or **RT_DRIVER**

23.2 Compiling the UEFI Driver

Before a new UEFI driver can be compiled, the Build Description (**.DSC**) for one or more build tips needs to be updated to compile the new UEFI driver. Each build produces at least 2 directories when they are built. One of the directories is the Tools directory which contains all of the compiled build tools. The second directory is the target directory for the EFI files. The directory name will signify what environment the inner files are compiled for. For example the X64 build will produce (in addition to the tools directory) an IA-32 directory with the IA-32 compiled files and an X64 with the X64 compiled files.

Table 39 lists the standard build tips that are the fully integrated EFI builds. The build tips include the UEFI core that produces the EFI Boot Services and EFI Runtime Services as well as a number of UEFI drivers that provide access to a wide variety of boot devices. These tips are used to produce native (except EBC) EFI driver images for soft loading or integration into an OptionROM.

Table 39. Build Tips Integrated into EFI Builds

Build Tips	Description
IPF	Produces the EFI firmware component for an Itanium-based platform.
Nt32	Produces a 32-bit Windows application that is an accurate emulation of the EFI execution environment and an IA-32 firmware image.
X64	Produces an Intel-64 (x64) firmware image.
EBC	Produces EBC images, but no firmware image.

It is also possible to create additional platform build tips in the EDK. To do this, add a new platform under the \Sample\Platform\ in the EDK and add all required makefile, DSC, .INF, and .ENV files.

Once a build tip is selected, the **DSC** in that build tip must be modified. All driver INFs must be referenced in the **[Components]** section. Example 188 shows part of the **[Components]** section before and after the XYZ UEFI driver was added.

```
Sample\Bus\Usb\UsbBus\Dxe\UsbBus.inf
Sample\Bus\Usb\UsbCbi\Dxe\Cbi0\UsbCbi0.inf
Sample\Bus\Usb\UsbCbi\Dxe\Cbi1\UsbCbi1.inf
Sample\Bus\Usb\UsbKb\Dxe\UsbKb.inf
Sample\Bus\Usb\UsbMassStorage\Dxe\UsbMassStorage.inf
Sample\Bus\Usb\UsbMouse\Dxe\UsbMouse.inf


Sample\Bus\Usb\UsbBus\Dxe\UsbBus.inf
Sample\Bus\Usb\UsbCbi\Dxe\Cbi0\UsbCbi0.inf
Sample\Bus\Usb\UsbCbi\Dxe\Cbi1\UsbCbi1.inf
Sample\Bus\Usb\UsbKb\Dxe\UsbKb.inf
Sample\Bus\Usb\UsbMassStorage\Dxe\UsbMassStorage.inf
Sample\Bus\Usb\UsbMouse\Dxe\UsbMouse.inf
Sample\Bus\PCI\XYZUefi\Make.inf
```

Example 188. Adding an UEFI driver to a Makefile

Once an UEFI driver has been added to the **DSC**, the UEFI driver can be built by executing the **nmake** command from the command line. See the release notes from the EDK for details. If the build completes, then the UEFI driver executable will be placed in the target directory below the build tip.

23.3 Integrating an UEFI driver into a firmware image

If an UEFI driver is being added to a native firmware image, direct support from the company supplying firmware image is required. Please work directly with the appropriate company.

23.4 Tools in the EDK

Please see documentation that is part of the *EDK* project for information on the build tools and stand alone tools that are part of the *EDK*.

Tools that are commonly used for Driver Development are covered in section 26.

24 Testing and Debugging UEFI drivers

When possible provide a driver in both native-instruction-set and EBC binary forms. Providing both of these forms allows the OEM firmware to simulate testing the driver in a fast best-case scenario and a slower scenario. If the driver is tested to work as both an EBC and native-instruction-set binary, it is expected that there will be fewer timing sensitivities to the driver and it will be more robust.

24.1 EFI Shell Debugging

There are several UEFI Shell commands that can be used to help debug UEFI drivers. These UEFI Shell commands are fully documented in the *EFI Shell Users Guide*, so the full capabilities of the UEFI Shell commands will not be discussed here. The UEFI Shell that is included in the *EDK* is a reference implementation of an UEFI Shell that may be customized for various platforms. As a result, the UEFI Shell commands described here may not behave identically on all platforms. There is also a built-in UEFI Shell command call `help` that will provide a detailed description of an UEFI Shell command. Table 40 lists Shell commands that can be used to test and debug UEFI drivers along with the protocol and/or service exercised.

24.1.1 Testing Specific Protocols

Table 40. EFI Shell Commands

Command	Protocol Tested	Service Tested
Load -nc		Driver entry point. Supported()
Load	 Driver Binding Driver Binding	Driver entry point. Supported() Start()
Unload	Loaded Image	Unload()
Connect	Driver Binding Driver Binding	Supported() Start()
Disconnect	Driver Binding	Stop()

Reconnect	Driver Binding Driver Binding Driver Binding	Supported() Start() Stop()
Drivers	Component Name	GetDriverName()
Devices	Component Name Component Name	GetDriverName() GetControllerName()
DevTree	Component Name	GetControllerName()
Dh -d	Component Name Component Name	GetDriverName() GetControllerName()
DrvCfg -s	Driver Configuration	SetOptions()
DrvCfg -f	Driver Configuration	ForceDefaults()
DrvCfg -v	Driver Configuration	OptionsValid()
DrvDiag	Driver Diagnostics	RunDiagnostics()

24.1.2 Other Shell testing

There are other tests that can be performed from within the EFI shell. These are not testing a specific protocol, but are testing for other coding practices.

Table 41 Other Shell Testing Procedure

Shell Command sequence	What it tests
Shell> Memmap **Shell> Dh** **Shell> Load DriverName.efi** **Shell> Memmap** **Shell> Dh** **Shell> Unload DriverHandle** **Shell> Memmap** **Shell> Dh**	Tests for incorrectly matched up **DriverEntryPoint** and **Unload()** functions. This will catch memory allocation that is not unallocated, protocols that are installed and not uninstalled, etc…

Shell> Memmap **Shell> Connect DeviceHandle DriverHandle** **Shell> Memmap** **Shell> Disconnect DeviceHandle DriverHandle** **Shell> Memmap** **Shell> Reconnect DeviceHandle** **Shell> Memmap**	Tests for incorrectly matched up DriverBinding **Start()** and **Stop()** functions. This will catch memory allocation that is not unallocated.
Shell> dh **Shell> Connect DeviceHandle DriverHandle** **Shell> dh** **Shell> Disconnect DeviceHandle DriverHandle** **Shell> dh** **Shell> Reconnect DeviceHandle** **Shell> dh**	Tests for incorrectly matched up DriverBinding **Start()** and **Stop()** functions. This will catch protocols that are installed and not uninstalled.
Shell> OpenInfo DeviceHandle **Shell> Connect DeviceHandle DriverHandle** **Shell> OpenInfo DeviceHandle** **Shell> Disconnect DeviceHandle DriverHandle** **Shell> OpenInfo DeviceHandle** **Shell> Reconnect DeviceHandle** **Shell> OpenInfo DeviceHandle**	Tests for incorrectly matched up DriverBinding **Start()** and **Stop()** functions. This will catch protocols that are opened and not closed.

24.1.3 Loading UEFI drivers

Table 42 lists the two UEFI Shell commands that are available to load and start UEFI drivers.

Table 42. EFI Shell Commands for Loading UEFI drivers

Command	Description
Load	Loads an UEFI driver from a file. UEFI driver files typically have an extension of **.efi**. The **Load** command has one important option, the **-nc** ("No Connect") option, for UEFI driver developers. When the **Load** command is used without the **-nc** option, then the loaded driver will automatically be connected to any devices in the system that it is able to manage. This means that the UEFI driver's entry point is executed and then the EFI Boot Service **ConnectController()** is called. If the UEFI driver produces the Driver Binding Protocol in the driver's entry point, then the **ConnectController()** call will exercise the **Supported()** and **Start()** services of Driver Binding Protocol that was produced. If the **-nc** option is used with the **Load** command, then this automatic connect operation will not be performed. Instead, only the UEFI driver's entry point is executed. When the **-nc** option is used, the UEFI Shell command **Connect** can be used to connect the UEFI driver to any devices in the system that it is able to manage. The **Load** command can also take wild cards, so multiple UEFI drivers can be loaded at the same time. The code below shows the following examples of the **Load** command: Example 1: Loads and does not connect the UEFI driver image **EfiDriver.efi**. This example exercises only the UEFI driver's entry point. Example 2: Loads and connects the UEFI driver image called **EfiDriver.efi**. This example exercises the UEFI driver's entry point and the **Supported()** and **Start()** functions of the Driver Binding Protocol. Example 3: Loads and connects all the UEFI drivers with an **.efi** extension from **fs0:**, exercising the UEFI driver entry points and their **Supported()** and **Start()** functions of the Driver Binding Protocol. **fs0:> Load -nc EfiDriver.efi** **fs0:> Load EfiDriver.efi** **fs0:> Load *.efi**
LoadPciRom	Simulates the load of a PCI option ROM by the PCI bus driver. This command parses a ROM image that was produced with the **EfiRom** build utility. Details on the **EfiRom** build utility can be found in 26.1. The **LoadPciRom** command will find all the UEFI drivers in the ROM image and will attempt to load and start all the UEFI drivers. This command helps test the ROM image before it is burned into a PCI adapter's ROM. No automatic connects are performed by this command, so only the UEFI driver's entry point will be exercised by this command. The UEFI Shell command **Connect** will have to be used for the loaded UEFI drivers to start managing devices. The example below loads and calls the entry point of all the UEFI drivers in the ROM file called **MyAdapter.ROM**. **fs0:> LoadPciRom MyAdapter.ROM**

24.1.4 Unloading UEFI drivers

Table 43 lists the UEFI Shell commands that can be used to unload an UEFI driver if it is unloadable.

Table 43. EFI Shell Commands for Unloading UEFI drivers

Command	Description
Unload	Unloads an UEFI driver if it is unloadable. This command takes a single argument that is the image handle number of the UEFI driver to unload. The **`Dh -p Image`** command and the **`Drivers`** command can be used to search for the image handle of the driver to unload. Once the image handle number is known, an unload operation can be attempted. The **`Unload`** command may fail for one of the following two reasons: The UEFI driver may not be unloadable, because UEFI drivers are not required to be unloadable. The UEFI driver might be unloadable, but it may not be able to be unloaded right now. Some UEFI drivers may need to be disconnected before they are unloaded. They can be disconnected with the **`Disconnect`** command. The following example unloads the UEFI driver on handle 27. If the UEFI driver on handle 27 is unloadable, it will have registered an **`Unload()`** function in its Loaded Image Protocol. This command will exercise the UEFI driver's **`Unload()`** function. **`Shell> Unload 27`**

24.1.5 Connecting UEFI drivers

Table 44 lists the three UEFI Shell commands that can be used to test the connecting of UEFI drivers to devices. There are many options for using these commands. A few are shown in Table 44.

Table 44. EFI Shell Commands for Connecting UEFI drivers

Command	Description
Connect	Can be used to connect all UEFI drivers to all devices in the system or connect UEFI drivers to a single device. The code below shows the following examples of the **`Connect`** command: Example 1: Connects all drivers to all devices. Example 2: Connects all drivers to the device that is abstracted by handle 23. Example 3: Connects the UEFI driver on handle 27 to the device that is abstracted by handle 23. **`fs0:> Connect -r`** **`fs0:> Connect 23`** **`fs0:> Connect 23 27`**

Disconnect	Stops UEFI drivers from managing a device. The code below shows the following examples of the **Disconnect** command: Example 1: Disconnects all drivers from all devices. However, the use of this command is not recommended, because it will also disconnect all the console devices. Example 2: Disconnects all the UEFI drivers from the device represented by handle 23. Example 3: Disconnects the UEFI driver on handle 27 from the device represented by handle 23. Example 1: Destroys the child represented by handle 32. The UEFI driver on handle 27 produced the child when it started managing the device on handle 23. **fs0:> Disconnect -r** **fs0:> Disconnect 23** **fs0:> Disconnect 23 27** **fs0:> Disconnect 23 27 32**
Reconnect	Is the equivalent of executing the **Disconnect** and **Connect** commands back to back. The **Reconnect** command is the best command for testing the Driver Binding Protocol of UEFI drivers. This command tests the **Supported()**, **Start()**, and **Stop()** services of the Driver Binding Protocol. The **Reconnect -r** command tests the Driver Binding Protocol for every UEFI driver that follows the UEFI Driver Model. Use this command before an UEFI driver is loaded to verify that the current set of drivers pass the **Reconnect -r** test, and then load the new UEFI driver and rerun the **Reconnect -r** test. An UEFI driver is not complete until it passes this interoperability test with the UEFI core and the full set of UEFI drivers. The code below shows the following examples of the **Reconnect** command: Example 1: Reconnects all the UEFI drivers to the device handle 23. Example 2: Reconnects the UEFI driver on handle 27 to the device on handle 23. Example 3: Reconnects all the UEFI drivers in the system. **fs0:> Reconnect 23** **fs0:> Reconnect 23 27** **fs0:> Reconnect -r**

24.1.6 Driver and Device Information

Table 45 lists the UEFI Shell commands that can be used to dump information about the UEFI drivers that follow the UEFI Driver Model. Each of these commands shows information from a slightly different perspective.

Table 45. EFI Shell Commands for Driver and Device Information

Command	Description
Drivers	Lists all the UEFI drivers that follow the UEFI Driver Model. It uses the **GetDriverName()** service of the Component Name protocol to retrieve the human-readable name of each UEFI driver if it is available. It also shows the file path from which the UEFI driver was loaded. As UEFI drivers are loaded with the **Load** command, they will appear in the list of drivers produced by the **Drivers** command. The **Drivers** command can also show the name of the UEFI driver in different languages. The code below shows the following examples of the **Drivers** command: Example 1: Shows the **Drivers** command being used to list the UEFI drivers in the default language. Example 2: Shows the driver names in Spanish. **fs0:> Drivers** **fs0:> Drivers -lspa**
Devices	Lists all the devices that are being managed or produced by UEFI drivers that follow the UEFI Driver Model. This command uses the **GetControllerName()** service of the Component Name protocol to retrieve the human-readable name of each device that is being managed or produced by UEFI drivers. If a human-readable name is not available, then the EFI device path is used. The code below shows the following examples of the **Devices** command: Example 1: Shows the **Devices** command being used to list the UEFI drivers in the default language. Example 2: Shows the device names in Spanish. **fs0:> Drivers** **fs0:> Drivers -lspa**
DevTree	Similar to the **Devices** command. Lists all the devices being managed by UEFI drivers that follow the UEFI Driver Model. This command uses the **GetControllerName()** service of the Component Name Protocol to retrieve the human-readable name of each device that is being managed or produced by UEFI drivers. If the human-readable name is not available, then the EFI device path is used. This command also visually shows the parent/child relationships between all of the devices by displaying them in a tree structure. The lower a device is in the tree of devices, the more the device name is indented. The code below shows the following examples of the **DevTree** command: Example 1: Displays the device tree with the device names in the default language. Example 2: Displays the device tree with the device names in Spanish. Example 3: Displays the device tree with the device names shown as EFI device paths. **fs0:> DevTree** **fs0:> DevTree -lspa** **fs0:> DevTree -d**

Dh -d	Provides a more detailed view of a single driver or a single device than the **Drivers**, **Devices**, and **DevTree** commands. If a driver binding handle is used with the **Dh -d** command, then a detailed description of that UEFI driver is provided along with the devices that the driver is managing and the child devices that the driver has produced. If a device handle is used with the **Dh -d** command, then a detailed description of that device is provided along with the drivers that are managing that device, that device's parent controllers, and the device's child controllers. If the **Dh -d** command is used without any parameters, then detailed information on all of the drivers and devices is displayed. The code below shows the following examples of the **Dh -d** command: Example 1: Displays the details on the UEFI driver on handle 27. Example 2: Displays the details for the device on handle 23. Example 3: Shows details on all the drivers and devices in the system. **fs0:> Dh -d 27** **fs0:> Dh -d 23** **fs0:> Dh -d**
OpenInfo	Provides detailed information on a device handle that is being managed by one or more UEFI drivers that follow the UEFI Driver Model. The **OpenInfo** command displays each protocol interface installed on the device handle, and the list of agents that have opened that protocol interface with the **OpenProtocol()** Boot Service. This command can be used in conjunction with the **Connect**, **Disconnect**, and **Reconnect** commands to verify that an UEFI driver is opening and closing protocol interfaces correctly. The following example shows the **OpenInfo** command being used to display the list of protocol interfaces on device handle 23 along with the list of agents that have opened those protocol interfaces. **fs0:> OpenInfo 23**

24.1.7 Testing the Driver Configuration Protocol

Table 46 lists the UEFI Shell commands that can be used to test the Driver Configuration Protocol.

Table 46. EFI Shell Commands for Testing the Driver Configuration Protocol

Command	Description
DrvCfg	Provides the services that are required to test the Driver Configuration Protocol implementation of an UEFI driver. This command can show all the devices that are being managed by UEFI drivers that support the Driver Configuration Protocol. The **Devices** and **Drivers** commands will also show the drivers that support the Driver Configuration Protocol and the devices that those drivers are managing or have produced. Once a device has been chosen, the **DrvCfg** command can be used to invoke the **SetOptions()**, **ForceDefaults()**, or **OptionsValid()** services of the Driver Configuration Protocol. The code below shows the following examples of the **DrvCfg** command: Example 1: Displays all the devices that are being managed by UEFI drivers that support the Driver Configuration Protocol. Example 2: Forces defaults on all the devices in the system. Example 3: Validates the options on all the devices in the system. Example 4: Invokes the **SetOptions()** service of the Driver Configuration Protocol for the driver on handle 27 and the device on handle 23. **fs0:> DrvCfg** **fs0:> DrvCfg -f** **fs0:> DrvCfg -v** **fs0:> DrvCfg -s 23 27**

24.1.8 Testing the Driver Diagnostics Protocol

Table 47 lists the UEFI Shell commands that can be used to test the Driver Diagnostics Protocol.

Table 47. EFI Shell Commands for Testing the Driver Diagnostics Protocol

Command	Description
DrvDiag	Provides the ability to test all the services of the Driver Diagnostics Protocol that is produced by an UEFI driver. This command is able to show the devices that are being managed by UEFI drivers that support the Driver Diagnostics Protocol. The **Devices** and **Drivers** commands will also show the drivers that support the Driver Diagnostics Protocol and the devices that those drivers are managing or have produced. Once a device has been chosen, the **DrvDiag** command can be used to invoke the **RunDiagnostics()** service of the Driver Diagnostics Protocol. The code below shows the following examples of the **DrvDiag** command: Example 1: Displays all the devices that are being managed by UEFI drivers that support the Driver Diagnostics Protocol. Example 2: Invokes the **RunDiagnostics()** service of the Driver Diagnostics Protocol in standard mode for the driver on handle 27 and the device on handle 23. Example 3: Invokes the **RunDiagnostics()** service of the Driver Diagnostics Protocol in manufacturing mode for the driver on handle 27 and the device on handle 23. **fs0:> DrvDiag** **fs0:> DrvDiag -s 23 27** **fs0:> DrvDiag -m 23 27**

24.2 Debugging Code Statements

Every module will have a debug (check) build and a clean build. The debug build will include code for debug that will not be included in normal clean production builds. A debug build is enabled when the identifier **EFI_DEBUG** exists. A clean build is defined as when the **EFI_DEBUG** identifier does not exist.

The following debug macros can be used to insert debug code into a checked build. This debug code can greatly reduce the amount of time it takes to root cause a bug. These macros are enabled only in a debug build, so they do not take up any executable space in the production build. Table 48 describes the debug macros that are available.

Table 48. Available Debug Macros

Debug Macro	Description
ASSERT (*Expression***)**	For check builds, if *Expression* evaluates to **FALSE**, a diagnostic message is printed and the program is aborted. Aborting a program is usually done via the **EFI_BREAKPOINT ()** macro. For clean builds, *Expression* does not exist in the program and no action is taken. Code that is required for normal program execution should never be placed inside an **ASSERT** macro, as the code will not exist in a production build.
ASSERT_EFI_ERROR (*Status***)**	For check builds, an assert is generated if **Status** is an error. This macro is equivalent to **ASSERT (!EFI_ERROR (Status))** but is easier to read.

DEBUG (*(ErrorLevel, String, …)*)	For check builds, **String** and its associated arguments will be printed if the **ErrorLevel** of the macro is active. See Table 49 below for a definition of the **ErrorLevel** values.
DEBUG_CODE (*Code*)	For check builds, **Code** is included in the build. **DEBUG_CODE (** is on its own line and indented like normal code. All the debug code follows on subsequent lines and is indented an extra level. The **)** is on the line following all the code and indented at the same level as **DEBUG_CODE (**.
EFI_BREAKPOINT ()	On a check build, inserts a break point into the code.
DEBUG_SET_MEM (*Address, Length*)	For a check build, initializes the memory starting at *Address* for **Length** bytes with the value **BAD_POINTER**. This initialization is done to enable debug of code that uses memory buffers that are not initialized.
CR (*Record, TYPE, Field, Signature*)	The containing record macro returns a pointer to *TYPE* when given the structure's *Field* name and a pointer to it (*Record*). The **CR** macro returns the *TYPE* pointer for check and production builds. For a check build, an **ASSERT ()** is generated if the *Signature* field of *TYPE* is not equal to the *Signature* in the **CR ()** macro.

The **ErrorLevel** parameter referenced in the **DEBUG()** macro allows an UEFI driver to assign a different error level to each debug message. Critical errors should always be sent to the standard error device. However, informational messages that are used only to debug a driver should be sent to the standard error device only if the user wants to see those specific types of messages. The UEFI Shell supports the **Err** command that allows the user to set the error level. The EFI Boot Maintenance Manager allows the user to enable and select a standard error device. It is recommended that a serial port be used as a standard error device during debug so the messages can be logged to a file with a terminal emulator. Table 49 contains the list of error levels that are supported in the *EFI Shell.* Other levels are usable, but not defined for a specific area.

Note: ***DEBUG*** *(ErrorLevel, String, …)* *is not universally supported. Some EFI compliant systems may not print out the message.*

Table 49. Error Levels

Mnemonic	Value	Description
EFI_D_INIT	0x00000001	Initialization messages
EFI_D_WARN	0x00000002	Warning messages
EFI_D_LOAD	0x00000004	Load events
EFI_D_FS	0x00000008	EFI file system messages
EFI_D_POOL	0x00000010	EFI pool allocation and free messages
EFI_D_PAGE	0x00000020	EFI page allocation and free messages
EFI_D_INFO	0x00000040	Informational messages
EFI_D_VARIABLE	0x00000100	EFI variable service messages
EFI_D_BM	0x00000400	UEFI boot manager messages
EFI_D_BLKIO	0x00001000	EFI Block I/O Protocol messages
EFI_D_NET	0x00004000	EFI Simple Network Protocol, PXE base code, BIS messages

EFI_D_UNDI	0x00010000	UNDI driver messages
EFI_D_LOADFILE	0x00020000	Load File Protocol messages
EFI_D_EVENT	0x00080000	EFI Event Services messages
EFI_D_ERROR	0x80000000	Critical error messages

24.3 POST Codes

If an UEFI driver is being developed that cannot make use of the **DEBUG()** and **ASSERT()** macros, then a different mechanism must be used to help in the debugging process. Under these conditions, it is usually sufficient to send a small amount of output to a device to indicate what portions of an UEFI driver have executed and where error conditions have been detected. A few possibilities are presented below, but many others are possible depending on the devices that may be available on a specific platform. It is important to note that these mechanisms are useful during driver development and debug, but they should never be present in production versions of UEFI drivers because these types of devices are not present on all platforms.

There are a couple of possibilities. The first is to use a POST card.

24.3.1 Post Card Debug

A POST card add in card adapter that displays the hex value of an 8-bit I/O write cycle to address 0x80 (and sometimes 0x81 also). If an UEFI driver can depend on the PCI Root Bridge I/O Protocol being present, then the driver can use the services of the PCI Root Bridge I/O Protocol to send an 8-bit I/O write cycle to address 0x80. A driver can also use the services of the PCI I/O Protocol to write to address 0x80, as long as the pass-through BAR value is used. Example 189 below shows how the PCI Root Bridge I/O and PCI I/O Protocols can be used to send a value to a POST card.

```
EFI_STATUS                          Status;
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL     *PciRootBridgeIo;
EFI_PCI_IO_PROTOCOL                 *PciIo;
UINT8                               Value;

Value = 0xAA;
Status = PciRootBridgeIo->Io.Write (
                                PciRootBridgeIo,
                                EfiPciWidthUint8,
                                0x80,
                                1,
                                &Value
                                );

Value = 0xAA;
Status = PciIo->Io.Write (
                    PciIo,
                    EfiPciIoWidthUint8,
                    EFI_PCI_IO_PASS_THROUGH_BAR,
                    0x80,
                    1,
                    &Value
                    );
```

Example 189. POST Code Examples

24.3.2 Text-mode VGA frame buffer

The next possibility is a text-mode VGA frame buffer. If a system initialized the text-mode VGA display by default before the UEFI driver executed, then the UEFI driver can make use of the PCI Root Bridge I/O or PCI I/O Protocols to write text characters to the text-mode VGA display directly. Example 190 shows how the PCI Root Bridge I/O and PCI I/O Protocols can be used to send the text message "ABCD" to the text-mode VGA frame buffer. Some systems do not have a VGA controller, so this solution will not work on all systems.

```
EFI_STATUS                          Status;
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL  *PciRootBridgeIo;
EFI_PCI_IO_PROTOCOL                 *PciIo;
UINT8                               *Value;

Value = {'A',0x0f,'B',0x0f,'C',0x0f,'D',0x0f};

Status = PciRootBridgeIo->Mem.Write (
                                PciRootBridgeIo,
                                EfiPciWidthUint8,
                                0xB8000,
                                8,
                                Value
                                );

Status = PciIo->Mem.Write (
                      PciIo,
                      EfiPciIoWidthUint8,
                      EFI_PCI_IO_PASS_THROUGH_BAR,
                      0xB8000,
                      8,
                      Value
                      );
```

Example 190. VGA Display Examples

24.4 Other Options

Another option is to use some type of byte-stream-based device. This device could include a UART or a SMBus, for example. Like the POST card, the idea is to use the services of the PCI Root Bridge I/O or PCI I/O Protocols to initialize and send characters to the byte-stream device.

24.4.1 Com Cable

Many EFI compliant implementations allow for the use of a COM cable to send debug information to another system. This allows the person to see debug code statements and other output from a separate system.

25 Distributing UEFI drivers

There are 2 main methods for distributing drivers. The first is in memory on the device itself such as PCI OptionROM. The second method is to store the **.EFI** file on some media that has a driver already working for it such as a disk drive.

25.1 Memory in Device

In order to store the EFI Driver for a device on the device first there must be some storage location available on the device with enough space available, The second requirement is that the bus driver for the interface type (USB Bus driver for USB device) needs to know how to get the information out of the media and load it.

This capability of the host controller driver is defined not by UEFI, but by the SIG representing the bus upon which the device is connected. Please refer to USB, or PCI, or other SIG documentation for formatting and storing of UEFI compliant drivers on those devices.

25.2 Media

An EFI driver may be loaded from any accessible media in the system. The main issues with media are: the media in question requires its own driver, the firmware may not load by default drivers from media, and that some EFI bus drivers may give precedence to drivers stored in memory in the device.

If the media requires its own driver, there is a new requirement that the other driver is found and works. For some simple devices such as floppy drives, that may be safe, but for other devices, like a SCSI drive, that may be less certain.

There is no requirement for EFI compliant implementations to load drivers from media. The capability is there, but the requirement is not.

The PCI Bus driver in many EFI compliant implementations always gives precedence to drivers in the device. This means that it may or may not find and use other drivers.

The result of all of these issues is that the safest way to distribute an UEFI compliant driver is in memory in the device itself.

26 Utilities

26.1 EfiRom

The **EfiRom** tool helps in the development of UEFI drivers for PCI adapters. Once an UEFI driver for a PCI adapter has been added to a build tip, it needs to be packaged into a PCI option ROM. UEFI drivers stored in PCI option ROMs are automatically loaded and executed by the PCI bus driver. The PCI driver model that is documented in chapter 13 of the *UEFI 2.0 Specification* provides the ability to mix legacy BIOS images and UEFI driver images for IA-32, IA-64, X64, and EBC. The **EfiRom** build tool allows any combination of these image types to be combined into a single PCI option ROM image. The final output of the **EfiRom** tool can be used with a PROM programmer or a flash update utility to reprogram the option ROM device on a PCI adapter. There are many options to the **EfiRom** tool, which are listed below in Table 50 for reference. Below the references are several examples showing how the **EfiRom** tool can be used to build many different types of PCI option ROM images.

Table 50. EFIROM Tool Switches

Switches	Description
-p	Provides verbose output from this tool.
-l	Do not automatically set the LAST bit in the PCIR data structure of the last image in the output file.
-v VendorId	A required switch that specifies the 16-bit vendor ID in the PCIR data structure for all images specified with the **-e** and **-ec** switches.
-d DeviceId	A required switch that specifies the 16-bit device ID in the PCIR data structure for all images specified with the **-e** and **-ec** switches.
-cc ClassCode	An optional switch that specifies the 24-bit class code in the PCIR data structure for all images specified with the **-e** and **-ec** switches.
-rev Revision	An optional switch that specifies the 16-bit revision in the PCIR data structure for all images specified with the **-e** and **-ec** switches.
-o OutFileName	An optional switch.
-e Filenames	Specifies the list of UEFI driver images that are to be added to the PCI option ROM image.
-ec Filenames	Specifies the list of UEFI driver images that are to be compressed and added to the PCI option ROM image.
-b Filenames	Specifies the list of binary images that are to be added to the PCI option ROM image. If the binary file is not the last image, then the LAST bit in the PCIR data structure will be cleared. If it is the last image, then the LAST bit in the PCIR data structure will be set. If the -l switch is specified, then the LAST bit will not be modified.

-dump FileName	Dumps the contents of a PCI option ROM image that follows the *PCI 2.2 Specification* for PCI option ROM construction.

Example 191 shows the following examples using the **EfiRom** tool:

Example 1:

Builds a PCI option ROM that contains a single uncompressed UEFI driver. The vendor ID is **0xABCD**, and the device ID is **0x1234**.

Example 2:

Same as example 1, but the UEFI driver is compressed, which reduces the size of the PCI option ROM.

Example 3:

Same as example 2, but an output file is specified. Examples 1 and 2 will generate the output file **EfiDriver.ROM**. Example 2 generates the output file **PciOptionRom.ROM**.

Example 4:

Puts two compressed UEFI drivers in a PCI option ROM.

Example 5:

Puts a binary file and a compressed UEFI driver in a PCI option ROM. This example is how a legacy option ROM image can be combined with an UEFI driver to produce an adapter that works on PC-AT class systems and EFI systems.

Example 6:

Shows how the class code and code revision fields can also be specified. By default, these fields are cleared by the **EfiRom** tool.

Example 7:

Shows how the PCI option ROM image generated in example 3 can be dumped.

```
(1) Efirom -v 0xABCD -d 0x1234 -e EfiDriver.efi

(2) Efirom -v 0xABCD -d 0x1234 -ec EfiDriver.efi

(3) Efirom -v 0xABCD -d 0x1234 -ec EfiDriver.efi -o PciOptionRom.ROM

(4) Efirom -v 0xABCD -d 0x1234 -ec EfiDriverA.efi EfiDriverB.efi

(5) Efirom -v 0xABCD -d 0x1234 -b Bios.bin -ec EfiDriver

(6) Efirom -v 0xABCD -d 0x1234 -cc 0x030000 -rev 0x0001 -b
Bios.bin -ec EfiDriver

(7) Efirom -dump PciOptionRom.ROM
```

Example 191. EFIROM Tool Examples

26.2 EFI Compress

The **EfiCompress** tool helps to compress a file by either UEFI compress algorithm or Framework compress algorithm. The UEFI compress algorithm is a mixture of LZ77 algorithm and Huffman coding, the details could be found in Appendix H of the *UEFI 2.0 Specification.* The Framework compress algorithm could be found in *EDK* (souce file **Edk\Sample\Tools\Source\Common\TianoCompress.c**).

There is only one option to **EfiCompress** tool, which is listed below in Table 51 for reference. Below the references are examples showing how the **EfiCompress** tool can be used to compress a file.

Table 51. EfiCompress Tool Switches

Switches	Description
-tCompressType	Optional compress algorithm (EFI | Tiano), case insensitive. If omitted, compress type specified ahead is used, default is EFI, e.g. EfiCompress a.in a.out -tTiano b.in b.out c.in c.out -tEFI d.in d.out a.in and d.in are compressed using EFI compress algorithm, b.in and c.in are compressed using Tiano compress algorithm

Example 1:

Compress a file named **A.in** to **A.out** with default algorithm, EFI.

Example 2:

Compress **A.in** to **A.out** with UEFI compress algorithm, and compress **B.in** to **B.out** with Framework compress algorithm.

```
(1) EfiCompress A.in A.out

(2) EfiCompress –tEFI A.in A.out -tTiano B.in B.out
```

27 Driver Checklist

The following are the requirements for making a good, compliant, and efficient UEFI driver model driver.

- Determine Driver Type
- Identify Consumed I/O Protocols
- Identify Produced I/O Protocols
- Identify EFI Driver Model Protocols
  - Component Name Protocol
  - Driver Binding Protocol
    - Produce the fewest possible children in Driver Binding **`Start()`** function.
  - Driver Configuration Protocol
  - Driver Diagnostics Protocol
- Identify Additional Driver Features
  - Install an **`Unload()`** handler
- Identify Target Platforms
  - Native IA-32
  - Native IA-64
  - Native x64
  - EFI Byte Code
- Use EFI Compression to reduce the size of the EfiRom
- Test the driver
  - Test all produced protocols
  - Test on multiple platforms
  - Pass all UEFI SCT tests for your device
- For Pci Drivers
  - Always Call **`PciIo->Attributes()`**
    - Advertises Dual Address Cycle Capability
    - Save and Enable Attributes in **`Start()`**
    - Disable Attributes in **`Stop()`**
  - DMA – Bus Master Write Operations

- Must call **`PciIo->Flush()`**
- DMA – Setting Up with **`PciIo->Map(`**)
  - Do Not Use Returned DeviceAddress
  - Not all chipsets have 1:1 bus/system mappings

Appendix A
EFI Data Types

Table 52 contains the set of base types that are used in all UEFI applications and UEFI drivers. These are the base types that should be used to build more complex unions and structures. The file **EFIBIND.H** in the *EDK* contains the code that is required to map compiler-specific data types to the EFI data types. If a new compiler is used, only this one file should be updated; all other EFI-related sources should compile unmodified. Table 53 contains the modifiers that can be used in conjunction with the EFI data types.

Table 52. Common EFI Data Types

Mnemonic	Description
BOOLEAN	Logical Boolean. 1-byte value containing a 0 for **FALSE** or a 1 for **TRUE**. Other values are undefined.
INTN	Signed value of native width. (4 bytes on IA-32, 8 bytes on Itanium architecture operations)
UINTN	Unsigned value of native width. (4 bytes on IA-32, 8 bytes on Itanium architecture operations)
INT8	1-byte signed value.
UINT8	1-byte unsigned value.
INT16	2-byte signed value.
UINT16	2-byte unsigned value.
INT32	4-byte signed value.
UINT32	4-byte unsigned value.
INT64	8-byte signed value.
UINT64	8-byte unsigned value.
CHAR8	1-byte character.
CHAR16	2-byte character. Unless otherwise specified, all strings are stored in the UTF-16 encoding format as defined by Unicode 2.1 and ISO/IEC 10646 standards.
VOID	Undeclared type.
EFI_GUID	128-bit buffer containing a unique identifier value. Unless otherwise specified, aligned on a 64-bit boundary.
EFI_STATUS	Status code. Type INTN.
EFI_HANDLE	A collection of related interfaces. Type VOID *.
EFI_EVENT	Handle to an event structure. Type VOID *.
EFI_LBA	Logical block address. Type UINT64.

EFI_TPL	Task priority level. Type UINTN.
EFI_MAC_ADDRESS	32-byte buffer containing a network Media Access Controller address.
EFI_Ipv4_ADDRESS	4-byte buffer. An IPv4 internet protocol address.
EFI_Ipv6_ADDRESS	16-byte buffer. An IPv6 internet protocol address.
EFI_IP_ADDRESS	16-byte buffer aligned on a 4-byte boundary. An IPv4 or IPv6 internet protocol address.
<Enumerated Type>	Element of an enumeration. Type INTN.

Table 53. Modifiers for Common EFI Data Types

Mnemonic	Description
IN	Datum is passed to the function.
OUT	Datum is returned from the function.
OPTIONAL	Datum that is passed to the function is optional, and a **NULL** may be passed if the value is not supplied.
STATIC	The function has local scope. This modifier replaces the standard C static key word, so it can be overloaded for debugging.
VOLATILE	Declares a variable to be volatile and thus exempt from optimization to remove redundant or unneeded accesses. Any variable that represents a hardware device should be declared as **VOLATILE**.
CONST	Declares a variable to be of type **const**. This modifier is a hint to the compiler to enable optimization and stronger type checking at compile time.

Appendix B
EFI Status Codes

Table 54 contains the set of status code values that may be returned by EFI Boot Services, EFI Runtime Services, and UEFI protocol services. These status codes are defined in detail in Appendix D of the *UEFI 2.0 Specification*.

Table 54. EFI_STATUS Codes

Mnemonic	32-bit Architecture	64-bit Architecture
EFI_SUCCESS	0x00000000	0x0000000000000000
EFI_LOAD_ERROR	0x80000001	0x8000000000000001
EFI_INVALID_PARAMETER	0x80000002	0x8000000000000002
EFI_UNSUPPORTED	0x80000003	0x8000000000000003
EFI_BAD_BUFFER_SIZE	0x80000004	0x8000000000000004
EFI_BUFFER_TOO_SMALL	0x80000005	0x8000000000000005
EFI_NOT_READY	0x80000006	0x8000000000000006
EFI_DEVICE_ERROR	0x80000007	0x8000000000000007
EFI_WRITE_PROTECTED	0x80000008	0x8000000000000008
EFI_OUT_OF_RESOURCES	0x80000009	0x8000000000000009
EFI_VOLUME_CORRUPTED	0x8000000A	0x800000000000000A
EFI_VOLUME_FULL	0x8000000B	0x800000000000000B
EFI_NO_MEDIA	0x8000000C	0x800000000000000C
EFI_MEDIA_CHANGED	0x8000000D	0x800000000000000D
EFI_NOT_FOUND	0x8000000E	0x800000000000000E
EFI_ACCESS_DENIED	0x8000000F	0x800000000000000F
EFI_NO_RESPONSE	0x80000010	0x8000000000000010
EFI_NO_MAPPING	0x80000011	0x8000000000000011
EFI_TIMEOUT	0x80000012	0x8000000000000012
EFI_NOT_STARTED	0x80000013	0x8000000000000013
EFI_ALREADY_STARTED	0x80000014	0x8000000000000014
EFI_ABORTED	0x80000015	0x8000000000000015
EFI_ICMP_ERROR	0x80000016	0x8000000000000016
EFI_TFTP_ERROR	0x80000017	0x8000000000000017
EFI_PROTOCOL_ERROR	0x80000018	0x8000000000000018

EFI_INCOMPATIBLE_VERSION	0x80000019	0x8000000000000019
EFI_SECURITY_VIOLATION	0x8000001A	0x800000000000001A
EFI_CRC_ERROR	0x8000001B	0x800000000000001B
EFI_END_OF_MEDIA	0x8000001C	0x800000000000001C
EIF_END_OF_FILE	0x8000001F	0x800000000000001F
EFI_WARN_UNKOWN_GLYPH	0x00000001	0x0000000000000001
EFI_WARN_DELETE_FAILURE	0x00000002	0x0000000000000002
EFI_WARN_WRITE_FAILURE	0x00000003	0x0000000000000003
EFI_WARN_BUFFER_TOO_SMALL	0x00000004	0x0000000000000004

Appendix C
Quick Reference Guide

This appendix contains a summary of the services and GUIDs that are available to UEFI drivers, including the following:

- EFI Boot Services
- EFI Runtime Services
- UEFI driver Library Services
- GUID variables
- Protocol variables
- Various protocol services

Some of the GUIDs and protocols listed here are not part of the *UEFI 2.0 Specification*. Instead, they are extensions that are included in the *EDK* or *the Framework*.

C.1 EFI Boot Services and EFI Runtime Services

The EFI Boot Services and EFI Runtime Services are listed in Table 55, Table 56, and Table 57. These three tables list the most commonly used services, the rarely used services, and the services that should not be used from UEFI drivers. Services labeled with type *gBS* are EFI Boot Services, and services labeled with type *gRT* are EFI Runtime Services. A detailed explanation of the services that are rarely used or should not be used by UEFI drivers is included in chapter 2. Chapters 6 and 7 of the *UEFI 2.0 Specification* contain detailed descriptions of all the EFI Boot Services and EFI Runtime Services.

Table 55. EFI Services That Are Commonly Used by UEFI drivers

```
gBS->AllocatePool()
gBS->FreePool()
gBS->AllocatePages()
gBS->FreePages()
gBS->SetMem()
gBS->CopyMem()
gBS->CreateEvent()
gBS->CreateEventEx()
gBS->CloseEvent()
gBS->SignalEvent()
gBS->SetTimer()
gBS->CheckEvent()
gBS->InstallMultipleProtocolInterfaces()
gBS->UninstallMultipleProtocolInterfaces()
gBS->LocateHandleBuffer()
gBS->LocateProtocol()
gBS->OpenProtocol()
gBS->CloseProtocol()
gBS->OpenProtocolInformation()
gBS->RaiseTPL()
gBS->RestoreTPL()
gBS->Stall()
```

Table 56. EFI Services That Are Rarely Used by UEFI drivers

```
gBS->ReinstallProtocolInterface()
gBS->LocateDevicePath()
gBS->ConnectController()
gBS->DisconnectController()
gBS->GetNextMonotonicCount()
gBS->LoadImage()
gBS->StartImage()
gBS->UnloadImage()
gBS->Exit()
gBS->InstallConfigurationTable()
gBS->CalculateCrc32()
gRT->ConvertPointer()
gRT->GetNextHighMonotonicCount()
gRT->GetTime()
gRT->SetVariable()
```

Table 57. EFI Services That Should Not Be Used by UEFI drivers

```
gBS->GetMemoryMap()
gBS->ExitBootServices()
gBS->InstallProtocolInterface()
gBS->UninstallProtocolInterface()
gBS->HandleProtocol()
gBS->LocateHandle()
gBS->RegisterProtocolNotify()
gBS->ProtocolsPerHandle()
gBS->WaitForEvent()
gBS->SetWatchDogTimer()
gRT->SetVirtualAddressMap()
gRT->GetNextVariableName()
gRT->SetTime()
gRT->GetWakeupTime()
gRT->SetWakeupTime()
gRT->ResetSystem()
```

C.2 UEFI Driver Library Services

There are 2 main function libraries available in the *EDK*: GlueLib and UEFI Driver Library.

C.2.1 GlueLib

The better of the 2 and the more recently developed is the GlueLib function library. The reasons that this library is better to use is for 2 main reasons. Firstly the functions are coded better and will perform better than their counterparts in the UEFI Driver Library. Secondly the functions in this library are identical to the functions in the EDK Release II, which means that transition to that build environment will be easier.

C.2.2 UEFI Driver Library

Table 58 lists some of the UEFI driver Library Services. These services simplify the implementation of UEFI drivers. Most of these services are layered on top of the EFI Boot Services and EFI Runtime Services. A detailed description of these library services can be found in the *UEFI driver Library Specification*.

Table 58. UEFI Driver Library Functions

UEFI Driver Library Functions		
Initialization Functions **EfiInitializeDriverLib()** **EfiLibInstallDriverBinding()** **EfiLibInstallAllDriverProtocols()**	Link List Functions **InitializeListHead()** **IsListEmpty()** **RemoveEntryList()** **InsertTailList()** **InsertHeadList()** **SwapListEntries()**	Memory Functions **EfiCopyMem()** **EfiSetMem()** **EfiZeroMem()** **EfiCompareMem()** **EfiLibAllocatePool()** **EfiLibAllocateZeroPool()**
Device Path Functions **EfiDevicePathInstance()** **EfiAppendDevicePath()** **EfiAppendDevicePathNode()** **EfiAppendDevicePathInstance()** **EfiFileDevicePath()** **EfiDevicePathSize()** **EfiDuplicateDevicePath()**	String Functions **EfiStrCpy()** **EfiStrLen()** **EfiStrSize()** **EfiStrCmp()** **EfiStrCat()** **EfiLibLookupUnicodeString()** **EfiLibAddUnicodeString()** **EfiLibFreeUnicodeStringTable()**	Spin Lock Functions **EfiInitializeLock()** **EfiAcquireLock()** **EfiAcquireLockOrFail()** **EfiReleaseLock()**

Math Functions	Miscellaneous Functions	
LShiftU64() **RShiftU64()** **MultU64x32()** **DivU64x32()**	**EfiCompareGuid()** **EfiLibCreateProtocolNotifyEvent()** **EfiLibGetSystemConfigurationTable()**	

Table 59 lists the macros that are available to UEFI drivers, and Table 60 lists the constants that are available to UEFI drivers. These macros and constants simplify the implementation of UEFI drivers.

Table 59. EFI Macros

Miscellaneous Macros	Linked List Macros	Critical Error Macros
EFI_DRIVER_ENTRY_POINT () **CR ()** **MEMORY_FENCE ()** **EFI_ERROR ()** **EFI_FIELD_OFFSET ()** **EFI_SIGNATURE_16 ()** **EFI_SIGNATURE_32 ()** **EFI_SIGNATURE_64 ()**	**INITIALIZE_LIST_HEAD ()** **IS_LIST_EMPTY ()** **REMOVE_ENTRY_LIST ()** **INSERT_TAIL_LIST ()** **INSERT_HEAD_LIST ()** **SWAP_LIST_ENTRIES ()**	**ASSERT ()** **ASSERT_EFI_ERROR ()** **EFI_BREAKPOINT ()** **ASSERT_LOCKED ()** **EFI_DEADLOOP ()**
Math Macros	Memory Macros	Debug Macros
EFI_MIN () **EFI_MAX ()**	**EFI_SIZE_TO_PAGES ()** **EFI_PAGES_TO_SIZE ()** **ALIGN_POINTER ()**	**DEBUG ()** **DEBUG_CODE ()** **DEBUG_SET_MEM ()**

Table 60. EFI Constants

Mnemonic	Description
TRUE	One.
FALSE	Zero.
NULL	**VOID** pointer to zero.

C.3 GUID Variables

Table 61 is a partial list of GUIDs that are available to UEFI drivers in the *EDK* (there may be more added at any time). The *Directory Name* heading is the name of the GUID directory that can be used with the **EFI_GUID_DEFINITION()** macro. The *GUID Variable Names* heading contains the names of the global variables that are available to the UEFI driver that uses the **EFI_GUID_DEFINITION()** macro for that directory. For example, assume the following statement is added to an UEFI driver.

```
#include EFI_GUID_DEFINITION(PcAnsi)
```

Then, the following variables of type **EFI_GUID** will be available to that UEFI driver:
gEfiPcAnsiGuid gEfiVT100Guid gEfiVT100PlusGuid gEfiVTUTF8Guid

Table 61. EFI GUID Variables

Directory Name	GUID Variable Names
AcpiDescription	**gEfiAcpiDescriptionGuid**
AlternateFvBlock	**gEfiAlternateFvBlockGuid**
Bmp	**gEfiDefaultBmpLogoGuid**
BootState	**gEfiBootStateGuid**
Capsule	**gEfiCapsuleVendorGuid**
CompatibleMemoryTested	**gEfiCompatibleMemoryTestedGuid**
ConsoleInDevice	**gEfiConsoleInDeviceGuid**
ConsoleOutDevice	**gEfiConsoleOutDeviceGuid**
EfiShell	**gEfiShellFileGuid and gEfiMiniShellFileGuid**
EventLegacyBios	**gEfiEventLegacyBootGuid**
FlashMapHob	**gEfiFlashMapHobGuid**
HotPlugDevice	**gEfiHotPlugDeviceGuid**
IoBaseHob	**gEfiIoBaseHobGuid**
MemoryTypeInformation	**gEfi MemoryTypeInformationGuid**
PciExpressBaseAddress	**gEfiPciExpressBaseAddressGuid**
PciHotplugDevice	**gEfiPciHotplugDeviceGuid**
PciOptionRomTable	**gEfiPciOptionRomTableGuid**
PrimaryConsoleInDevice	**gEfiPrimaryConsoleInDeviceGuid**
PrimaryConsoleOutDevice	**gEfiPrimaryConsoleOutDeviceGuid**
PrimaryStandardErrorDevice	**gEfiPrimaryStandardErrorDeviceGuid**
SalSystemTable	**gEfiSalSystemTableGuid**
SmBios	**gEfiSmbiosTableGuid**
StandardErrorDevice	**gEfiStandardErrorDeviceGuid**

The complete list of GUIDs available to drivers can be gleaned by checking in each of the header (.h) files is each of the subdirectories in each of the directories listed below in the *EDK*.

- \Foundation\Guid\
- \Foundation\Efi\Guid\
- \Foundation\Framework\Guid\

C.4 Protocol Variables

Table 62 is a list of the protocols that are available to UEFI drivers in the EDK. The *Directory Name* heading is the name of the protocol directory that can be used with the **EFI_PROTOCOL_DEFINITION()** macro. The *Protocol GUID Variable Name* heading contains the names of the global variables that are available to a driver that uses the **EFI_PROTOCOL_DEFINITION()** macro for that directory. For example, assume the following statement is added to an UEFI driver.

```
#include EFI_PROTOCOL_DEFINITION(PciIo)
```

Then, the following variable of type **EFI_GUID**, along with the definition of the **EFI_PCI_IO_PROTOCOL**, would be available to that UEFI driver:

```
gEfiPciIoProtocolGuid
```

Note: *This may be out of date as the EDK may be changed more frequently than this document.*

Table 62. Protocol Variables

Directory Name	Protocol GUID Variable Names	Where Protocol Defined	Deprecated
AcpiS3Save	**gEfiAcpiS3SaveGuid**	Intel Framework	
AcpiSupport	**gEfiAcpiSupportGuid**	Intel Framework	
Arp	**gEfiArpServiceBindingProtocolGuid**	UEFI	
	gEfiArpProtocolGuid	UEFI	
BIS	**gEfiBisProtocolGuid**	UEFI	
BlockIo	**gEfiBlockIoProtocolGuid**	UEFI	
BootSctiptSave	**gEfiBootScriptSaveGuid**	Intel Framework	
BusSpecificDriverOverride	**gEfiBusSpecificDriverOverrideProtocolGuid**	UEFI	
ConsoleControl	**gEfiConsoleControlProtocolGuid**	EDK	
ComponentName	**gEfiComponentNameProtocolGuid**	UEFI	
CpuIo	**gEfiCpuIoProtocolGuid**	Intel Framework	
CustomizedDecompress	**gEfiCustomizedDecompressProtocolGuid**	EDK	
DataHub	**gEfiDataHubProtocolGuid**	Intel Framework	
DebugPort	**gEfiDebugPortProtocolGuid**	UEFI	
DebugAssert	**gEfiDebugAssertProtocolGuid**	EDK	

DebugMask	**gEfiDebugMaskProtocolGuid**	EDK	
DebugSupport	**gEfiDebugSupportProtocolGuid**	UEFI	
Decompress	**gEfiDecompressProtocolGuid**	UEFI	
DeviceIo	**gEfiDeviceIoProtocolGuid**	EFI 1.10	UEFI 2.0
DevicePath	**gEfiDevicePathProtocolGuid**	UEFI	
DevicePathFromText	**gEfiDevicePathFromTextProtocolGuid**	UEFI	
DevicePathToText	**gEfiDevicePathToTextProtocolGuid**	UEFI	
DevicePathUtilities	**gEfiDevicePathUtilitiesProtocolGuid**	UEFI	
Dhcp4	**gEfiDhcp4ServiceBindingProtocolGuid**	UEFI	
	gEfiDhcp4ProtocolGuid	UEFI	
DiskInfo	**gEfiDiskInfoProtocolGuid**	EDK	
	gEfiDiskInfoIdeInterfaceGuid	EDK	
	gEfiDiskInfoScsiInterfaceGuid	EDK	
	gEfiDiskInfoUsbInterfaceGuid	EDK	
DiskIo	**gEfiDiskIoProtocolGuid**	UEFI	
DriverBinding	**gEfiDriverBindingProtocolGuid**	UEFI	
DriverConfiguration	**gEfiDriverConfigurationProtocolGuid**	UEFI	UEFI 2.1
DriverDiagnostics	**gEfiDriverDiagnosticsProtocolGuid**	UEFI	
EBC	**gEfiEbcProtocolGuid**	UEFI	
EdidActive	**gEfiEdidActiveProtocolGuid**	UEFI	
EdidDiscovered	**gEfiEdidDiscoveredProtocolGuid**	UEFI	
EdidOverride	**gEfiEdidOverrideProtocolGuid**	UEFI	
EfiNetworkInterfaceIdentifier	**gEfiNetworkInterfaceIdentifierProtocolGuid**	UEFI	
	gEfiNetworkInterfaceIdentifierProtocolGuid_31	UEFI	
EfiOEMBadging	**gEfiOEMBadgingProtocolGuid**	EDK	
ExtendedSalBootService	**gEfiExtendedSalBootServiceProtocolGuid**	EDK	
ExtendedSalGuid	See the file for GUIDs	EDK	
FaultTolerantWriteLite	**gEfiFaultTolerantWriteLiteProtocolGuid**	EDK	
FileInfo	**gEfiFileInfoGuid**	UEFI	

FileSystemInfo	**gEfiFileSystemInfoGuid**	UEFI	
FileSystemVolumeLabelInfo	**gEfiFileSystemVolumeLabelInfoIdGuid**	UEFI	
FirmwareVolume	**gEfiFirmwareVolumeProtocolGuid**	Intel Framework	
FirmwareVolumeBlock	**gEfiFirmwareVolumeBlockProtocolGuid**	Intel Framework	
FirmwareVolumeDispatch	**gEfiFirmwareVolumeDispatchProtocolGuid**	EDK	
FormBrowser	**gEfiFormBrowserProtocolGuid**	Intel Framework	
FormCallback	**gEfiFormCallbackProtocolGuid**	Intel Framework	
FvbExtendion	**gEfiFvbExtensionProtocolGuid**	EDK	
GenericMemoryTest	**gEfiGenericMemTestProtocolGuid**		
GraphicsOutput	**gEfiGraphicsOutputProtocolGuid**	UEFI	
GuidedSectionExtrantion	**gEfiCrc32GuidedSectionExtractionProtocolGuid**		
Hash	**gEfiHashServiceBindingProtocolGuid**	UEFI	
	gEfiHashProtocolGuid	UEFI	
	gEfiHashAlgorithmSha1Guid	UEFI	
	gEfiHashAlgorithmSha224Guid	UEFI	
	gEfiHashAlgorithmSha256Guid	UEFI	
	gEfiHashAlgorithmSha384Guid	UEFI	
	gEfiHashAlgorithmSha512Guid	UEFI	
	gEfiHashAlgorithmMD5Guid	UEFI	
Hii	**gEfiHiiProtocolGuid**	Intel Framework	
IdeControllerInit	**gEfiIdeControllerInitProtocolGuid**	Intel Framework	
IncompatiblePciDeviceSupport	**gEfiIncompatiblePciDeviceSupportProtocolGuid**	Intel Framework	
Ip4	**gEfiIp4ServiceBindingProtocolGuid**	UEFI	
	gEfiIp4ProtocolGuid	UEFI	
Ip4Config	**gEfiIp4ConfigProtocolGuid**	UEFI	
IScsiInitiatorName	**gEfiIScsiInitiatorNameProtocolGuid**	UEFI	
IsaAcpi	**gEfiIsaAcpiProtocolGuid**	EDK	
IsaIo	**gEfiIsaIoProtocolGuid**	EDK	

Legacy8259	**gEfiLegacy8259ProtocolGuid**	Intel Framework	
LegacyBios	**gEfiLegacyBiosProtocolGuid**	Intel Framework	
LegacyBiosPlatform	**gEfiLegacyBiosPlatformProtocolGuid**	Intel Framework	
LegacyBiosThunk	**gEfiLegacyBiosThunkProtocolGuid**	EDK	
LegacyInterrupt	**gEfiLegacyInterruptProtocolGuid**	Intel Framework	
LegacyRegion	**gEfiLegacyRegionProtocolGuid**	Intel Framework	
LoadedImage	**gEfiLoadedImageProtocolGuid**	UEFI	
LoadFile	**gEfiLoadFileProtocolGuid**	UEFI	
LoadPe32Image	**gEfiLoadPeImageGuid**	EDK	
ManagedNetwork	**gEfiManagedNetworkServiceBindingProtocolGuid**	UEFI	
	gEfiManagedNetworkProtocolGuid	UEFI	
Mtftp4	**gEfiMtftp4ServiceBindingProtocolGuid**	Intel Framework	
	gEfiMtftp4ProtocolGuid	Intel Framework	
NicIp4Config	**gEfiNicIp4ConfigProtocolGuid**	EDK	
	gEfiNicIp4ConfigVariableGuid	EDK	
PciHostBridgeResourceAllocation	**gEfiPciHostBridgeResourceAllocationProtocolGuid**	Intel Framework	
PciHotPlugInit	**gEfiPciHotPlugInitProtocolGuid**	Intel Framework	
PciHotPlugRequest	**gEfiPciHotPlugRequestProtocolGuid**	EDK	
PciIo	**GEfiPciIoProtocolGuid**	UEFI	
PciPlatform	**gEfiPciPlatformProtocolGuid**	Intel Framework	
PciRootBridgeIo	**gEfiPciRootBridgeIoProtocolGuid**	UEFI	
Performance	**gEfiPerformanceProtocolGuid**	EDK	
PlatformDriverOverride	**gEfiPlatformDriverOverrideProtocolGuid**	UEFI	
PlatformMemTest	**gEfiPlatformMemTestGuid**	EDK	
Print	**gEfiPrintProtocolGuid**	EDK	
Ps2Policy	**gEfiPs2PolicyProtocolGuid**	EDK	
PxeBaseCode	**gEfiPxeBaseCodeProtocolGuid**	UEFI	

PxeBaseCodeCallBack	**gEfiPxeBaseCodeCallbackProtocolGuid**	UEFI	
PxeDhcp4	**gEfiPxeDhcp4ProtocolGuid**	EDK	
PxeDhcp4Callback	**gEfiPxeDhcp4CallbackProtocolGuid**	EDK	
ScsiIo	**gEfiScsiIoProtocolGuid**	UEFI 2.0	
ScsiPassThru	**gEfiScsiPassThruProtocolGuid**	EFI 1.10	UEFI 2.0
ScsiPassThruExt	**gEfiExtScsiPassThruProtocolGuid**	UEFI 2.0	
SectionExtraction	**gEfiSectionExtractionProtocolGuid**	Intel Framework	
SecurityPolicy	**gEfiSecurityPolicyProtocolGuid**	Intel Framework	
SerialIo	**gEfiSerialIoProtocolGuid**	UEFI	
SimpleFileSystem	**gEfiSimpleFileSystemProtocolGuid**	UEFI	
SimpleNetwork	**GEfiSimpleNetworkProtocolGuid**	UEFI	
SimplePointer	**gEfiSimplePointerProtocolGuid**	UEFI	
SimpleTextIn	**gEfiSimpleTextInProtocolGuid**	UEFI	
SimpleTextOut	**gEfiSimpleTextOutProtocolGuid**	UEFI	
Smbus	**gEfiSmbusProtocolGuid**	Intel Framework	
SmmAccess	**gEfiSmmAccessProtocolGuid**	Intel Framework	
SmmBase	**gEfiSmmBaseProtocolGuid**	Intel Framework	
	gEfiSmmCpuIoGuid	Intel Framework	
	gEfiSmmCommunicateHeaderGuid	Intel Framework	
SmmControl	**gEfiSmmControlProtocolGuid**	Intel Framework	
SmmCpuState	**gEfiSmmCpuSaveStateProtocolGuid**	Intel Framework	
SmmGpiDispatch	**gEfiSmmGpiDispatchProtocolGuid**	Intel Framework	
SmmIchnDispatch	**gEfiSmmIchnDispatchProtocolGuid**	Intel Framework	
SmmPeriodicTimerDispatch	**gEfiSmmPeriodicTimerDispatchProtocolGuid**	Intel Framework	
SmmPowerButtonDispatch	**gEfiSmmPowerButtonDispatchProtocolGuid**	Intel Framework	
SmmStandbyButtonDispatch	**gEfiSmmStandbyButtonDispatchProtocolGuid**	Intel Framework	

SmmStatusCode	**gEfiSmmStatusCodeProtocolGuid**	Intel Framework	
SmmSwDispatch	**gEfiSmmSwDispatchProtocolGuid**	Intel Framework	
SmmSxDispatch	**gEfiSmmSxDispatchProtocolGuid**	Intel Framework	
SmmUsbDispatch	**gEfiSmmUsbDispatchProtocolGuid**	Intel Framework	
TapeIo	**gEfiTapeIoProtocolGuid**	UEFI 2.0	
TcgService	**gEfiTcgProtocolGuid**	EDK	
Tcp	**gEfiTcpProtocolGuid**	EDK	
Tcp4	**gEfiTcp4ServiceBindingProtocolGuid**	UEFI	
	gEfiTcp4ProtocolGuid	UEFI	
TianoDecompress	**gEfiTianoDecompressProtocolGuid**	EDK	
Udp4	**gEfiUdp4ServiceBindingProtocolGuid**	UEFI	
	gEfiUdp4ProtocolGuid	UEFI	
UgaDraw	**gEfiUgaDrawProtocolGuid**	EFI 1.10	UEFI 2.0
UgaIo	**gEfiUgaIoProtocolGuid**	EFI 1.10	UEFI 2.0
UgaSplash	**gEfiUgaSplashProtocolGuid**	EDK	
UnicodeCollation	**gEfiUnicodeCollationProtocolGuid**	UEFI	
UsbAtapi	**gEfiUsbAtapiProtocolGuid**	EDK	
UsbHostController	**gEfiUsb2HcProtocolGuid**	UEFI 2.0	
	gEfiUsbHcProtocolGuid	EFI 1.10	UEFI 2.0
UsbIo	**gEfiUsbIoProtocolGuid**	UEFI	
VariableStore	**gEfiVariableStoreProtocolGuid**	EDK	
VgaMiniPort	**gEfiVgaMiniPortProtocolGuid**	EDK	
VirtualMemoryAccess	**gEfiVirtualMemoryAccessProtocolGuid**	EDK	
WinNtIo	**gEfiWinNtIoProtocolGuid**	EDK (NT32 build)	
	gEfiWinNtVirtualDisksGuid	EDK (NT32 build)	
	gEfiWinNtPhysicalDisksGuid	EDK (NT32 build)	
	gEfiWinNtFileSystemGuid	EDK (NT32 build)	
	gEfiWinNtSerialPortGuid	EDK (NT32 build)	

	gEfiWinNtConsoleGuid	EDK (NT32 build)	
WinNtThunk	**gefiWinNtThunkProtocolGuid**	EDK (NT32 build)	

A complete current list of Protocols available to drivers can be gleaned by checking in each of the header (.h) files is each of the subdirectories in each protocol directory listed below (from the *EDK*):

- \Foundation\Protocol\
- \Foundation\Efi\Protocol\
- \Foundation\Framework\Protocol\
- \Sample\Platform\<Build>\Protocol\ - for whichever build you are building. Only NT32 build had anything at the time of writing this.

C.5 Various Protocol Services

Table 63 contains a partial list of the list of the protocols that are available to UEFI drivers along with the list of services that each protocol provides. This table does not show any of the data elements that a protocol interface may contain. The *UEFI 2.0 Specification* should be referenced for additional details on each protocol. Each protocol is named by its C data structure. These protocols are available to UEFI drivers that use the **EFI_PROTOCOL_DEFINITION()** macro for a specific protocol. For example, if an UEFI driver intends to consume or produce the EFI USB I/O Protocol, it would need to include the following statement in its implementation: **#include EFI_PROTOCOL_DEFINITION(UsbIo)**.

Table 63. UEFI Protocol Service Summary

PROTOCOL	Service
EFI_BIS_PROTOCOL	Initialize() Shutdown() Free() GetBootObjectAuthorizationCertificate() GetBootObjectAuthorizationCheckFlag() GetBootObjectAuthorizationUpdateToken() GetSignatureInfo() UpdateBootObjectAuthorization() VerifyBootObject() VerifyObjectWithCredential()

EFI_BLOCK_IO_PROTOCOL	Reset() ReadBlocks() WriteBlocks() FlushBlocks()
EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL	GetDriver()
EFI_COMPONENT_NAME_PROTOCOL	GetDriverName() GetControllerName()
EFI_DEBUGPORT_PROTOCOL	Reset() Write() Read() Poll()
EFI_DEBUG_SUPPORT_PROTOCOL	GetMaximumProcessorIndex() RegisterPeriodicCallback() RegisterExceptionCallback() InvalidateInstructionCache()
EFI_DECOMPRESS_PROTOCOL	GetInfo() Decompress()
EFI_DEVICE_PATH_PROTOCOL	
EFI_DISK_IO_PROTOCOL	ReadDisk() WriteDisk()
EFI_DRIVER_BINDING_PROTOCOL	Supported() Start() Stop()
EFI_DRIVER_CONFIGURATION_PROTOCOL	SetOptions() OptionsValid() ForceDefaults()
EFI_DRIVER_DIAGNOSTICS_PROTOCOL	RunDiagnostics()
EFI_EBC_PROTOCOL	CreateThunk() UnloadImage() RegisterICacheFlush()

EFI_FILE_PROTOCOL	Open() Close() Delete() Read() Write() SetPosition() GetPosition() GetInfo() SetInfo() Flush()
EFI_GRAPHICS_OUTPUT_PROTOCOL	QueryMode() SetMode() Blt()
EFI_ISA_ACPI_PROTOCOL	DeviceEnumerate() SetPower() GetCurResource() GetPosResource() SetResource() EnableDevice() InitDevice() InterfaceInit()
EFI_ISA_IO_PROTOCOL	Mem.Read() Mem.Write() Io.Read() Io.Write() CopyMem() Map() Unmap() AllocateBuffer() FreeBuffer() Flush()
EFI_LOADED_IMAGE_PROTOCOL	Unload()
EFI_LOAD_FILE_PROTOCOL	LoadFile()

EFI_PCI_IO_PROTOCOL	PollMem() PollIo() Mem.Read() Mem.Write() Io.Read() Io.Write() Pci.Read() Pci.Write() CopyMem() Map() Unmap() AllocateBuffer() FreeBuffer() Flush() GetLocation() Attributes() GetBarAttributes() SetBarAttributes()
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL	PollMem() PollIo() Mem.Read() Mem.Write() Io.Read() Io.Write() Pci.Read() Pci.Write() CopyMem() Map() Unmap() AllocateBuffer() FreeBuffer() Flush() GetAttributes() SetAttributes() Configuration()
EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL	GetDriver() GetDriverPath() DriverLoaded()

EFI_PXE_BASE_CODE_PROTOCOL	Start() Stop() Dhcp() Discover() Mtftp() UdpWrite() UdpRead() SetIpFilter() Arp() SetParameters() SetStationIp() SetPackets()
EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL	Callback() PassThru() GetNextDevice() BuildDevicePath() GetTargetLun() ResetChannel() ResetTarget()
EFI_EXT_SCSI_PASS_THRU_PROTOCOL	Mode PassThru() GetNextTargetLun() BuildDevicePath() GetTargetLun() ResetChannel() ResetTargetLun() GetNextTarget()
EFI_SERIAL_IO_PROTOCOL	Reset() SetAttributes() SetControl() GetControl() Write() Read()
EFI_SIMPLE_FILE_SYSTEM_PROTOCOL	OpenVolume()

EFI_SIMPLE_NETWORK_PROTOCOL	Start() Stop() Initialize() Reset() Shutdown() ReceiveFilters() StationAddress() Statistics() MCastIPtoMAC() NvData() GetStatus() Transmit() Receive()
EFI_SIMPLE_POINTER_PROTOCOL	Reset() GetState()
EFI_SIMPLE_TEXT_INPUT_PROTOCOL	Reset() ReadKeyStroke()
EFI_SIMPLE_TEXT_OUT_PROTOCOL	Reset() OutputString() TestString() QueryMode() SetMode() SetAttribute() ClearScreen() SetCursorPosition() EnableCursor()
UNICODE_COLLATION_PROTOCOL	StriColl() MetaiMatch() StrLwr() StrUpr() FatToStr() StrToFat()
EFI_USB_ATAPI_PROTOCOL	UsbAtapiPacketCmd() UsbAtapiReset()

EFI_USB2_HC_PROTOCOL	GetCapability() Reset() GetState() SetState() ControlTransfer() BulkTransfer() AsyncInterruptTransfer() SyncInterruptTransfer() IsochronousTransfer() AsyncIsochronousTransfer() GetRootHubPortStatus() SetRootHubPortFeature() ClearRootHubPortFeature()
EFI_USB_IO_PROTOCOL	UsbControlTransfer() UsbBulkTransfer() UsbAsyncInterruptTransfer() UsbSyncInterruptTransfer() UsbIsochronousTransfer() UsbAsyncIsochronousTransfer() UsbGetDeviceDescriptor() UsbGetConfigDescriptor() UsbGetInterfaceDescriptor() UsbGetEndpointDescriptor() UsbGetStringDescriptor() UsbGetSupportedLanguages() UsbPortReset()

Table 64 contains the list of debug error level that can be used with the **DEBUG()** macros.

Table 64. Error Levels

Mnemonic	Value	Description
EFI_D_INIT	0x00000001	Initialization messages
EFI_D_WARN	0x00000002	Warning messages
EFI_D_LOAD	0x00000004	Load events
EFI_D_FS	0x00000008	EFI File System messages
EFI_D_POOL	0x00000010	EFI pool allocation and free messages
EFI_D_PAGE	0x00000020	EFI page allocation a free messages

EFI_D_INFO	0x00000040	Informational messages
EFI_D_VARIABLE	0x00000100	EFI variable service messages
EFI_D_BM	0x00000400	UEFI boot manager messages
EFI_D_BLKIO	0x00001000	EFI Block I/O Protocol messages
EFI_D_NET	0x00004000	EFI Simple Network Protocol, PXE Base Code, BIS messages
EFI_D_UNDI	0x00010000	UNDI driver messages
EFI_D_LOADFILE	0x00020000	Load File Protocol messages
EFI_D_EVENT	0x00080000	EFI Event Services messages
EFI_D_ERROR	0x80000000	Critical error messages

Appendix D
Disk I/O Protocol and Disk I/O Driver

This appendix contains the source files to the Disk I/O Protocol, the EFI Global Variable GUID, and the source files to the disk I/O driver. The disk I/O driver is a device driver that consumes the Block I/O Protocol and produces the Disk I/O Protocol on the same device handle.

The Disk I/O Protocol is composed of the two files **DiskIo.h** and **DiskIo.c**. **DiskIo.h** contains the GUID, function prototypes, and the data structure for the Disk I/O Protocol. The **DiskIo.c** file contains the global variable for the Disk I/O Protocol GUID.

The EFI Global Variable GUID is composed of the two files **GlobalVariable.h** and **GlobalVariable.c**. **GlobalVariable.h** contains the definition of the GUID that is used to access EFI environment variables with the UEFI variable services, and the **GlobalVariable.c** file contains the declaration of the global variable for GUID defined in **GlobalVariable.h**.

The disk I/O driver is composed of the three files **DiskIo.h**, **DiskIo.c**, and **ComponentName.c**. **DiskIo.h** contains the include statements for the EFI Boot Services, EFI Runtime Services, the UEFI driver Library, the list of protocols that the disk I/O driver consumes, the list of protocols that the disk I/O driver produces, the private context structure, and the declarations of the driver's global variable. The private context structure contains the Disk I/O Protocol that is installed onto the device handle and a set of private data fields. The **DiskIo.c** file contains the disk I/O driver's entry point, the implementation of the Driver Binding Protocol functions, and the implementation of the Disk I/O Protocol functions. The **ComponentName.c** file contains the implementation of the Component Name Protocol functions.

D.1 Disk I/O Protocol - DiskIo.h

```
/*++

Copyright (c) 2004, Intel Corporation
All rights reserved. This program and the accompanying materials
are licensed and made available under the terms and conditions of the BSD
License
which accompanies this distribution.  The full text of the license may be
found at
http://opensource.org/licenses/bsd-license.php

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.

Module Name:

  DiskIo.h

Abstract:

  Disk IO protocol as defined in the EFI 1.0 specification.

  The Disk IO protocol is used to convert block oriented devices into
byte
  oriented devices. The Disk IO protocol is intended to layer on top of
the
  Block IO protocol.

--*/

#ifndef __DISK_IO_H__
#define __DISK_IO_H__

#define EFI_DISK_IO_PROTOCOL_GUID \
  { \
    0xce345171, 0xba0b, 0x11d2, 0x8e, 0x4f, 0x0, 0xa0, 0xc9, 0x69, 0x72,
0x3b \
  }

EFI_FORWARD_DECLARATION (EFI_DISK_IO_PROTOCOL);

typedef
EFI_STATUS
(EFIAPI *EFI_DISK_READ) (
  IN EFI_DISK_IO_PROTOCOL         * This,
  IN UINT32                       MediaId,
  IN UINT64                       Offset,
  IN UINTN                        BufferSize,
  OUT VOID                        *Buffer
  )
/*++

  Routine Description:
    Read BufferSize bytes from Offset into Buffer.

  Arguments:
    This       - Protocol instance pointer.
    MediaId    - Id of the media, changes every time the media is
replaced.
    Offset     - The starting byte offset to read from
    BufferSize - Size of Buffer
    Buffer     - Buffer containing read data

  Returns:
    EFI_SUCCESS           - The data was read correctly from the device.
```

```
    EFI_DEVICE_ERROR      - The device reported an error while performing
the read.
    EFI_NO_MEDIA          - There is no media in the device.
    EFI_MEDIA_CHNAGED     - The MediaId does not matched the current
device.
    EFI_INVALID_PARAMETER - The read request contains device addresses
that are not
                            valid for the device.

--*/
;

typedef
EFI_STATUS
(EFIAPI *EFI_DISK_WRITE) (
  IN EFI_DISK_IO_PROTOCOL         * This,
  IN UINT32                       MediaId,
  IN UINT64                       Offset,
  IN UINTN                        BufferSize,
  IN VOID                         *Buffer
  )
/*++

  Routine Description:
    Read BufferSize bytes from Offset into Buffer.

  Arguments:
    This       - Protocol instance pointer.
    MediaId    - Id of the media, changes every time the media is
replaced.
    Offset     - The starting byte offset to read from
    BufferSize - Size of Buffer
    Buffer     - Buffer containing read data

  Returns:
    EFI_SUCCESS           - The data was written correctly to the device.
    EFI_WRITE_PROTECTED   - The device can not be written to.
    EFI_DEVICE_ERROR      - The device reported an error while performing
the write.
    EFI_NO_MEDIA          - There is no media in the device.
    EFI_MEDIA_CHNAGED     - The MediaId does not matched the current
device.
    EFI_INVALID_PARAMETER - The write request contains device addresses
that are not
                            valid for the device.

--*/
;

#define EFI_DISK_IO_PROTOCOL_REVISION 0x00010000

typedef struct _EFI_DISK_IO_PROTOCOL {
  UINT64          Revision;
  EFI_DISK_READ   ReadDisk;
  EFI_DISK_WRITE  WriteDisk;
} EFI_DISK_IO_PROTOCOL;

extern EFI_GUID gEfiDiskIoProtocolGuid;

#endif
```

Disk I/O Protocol - DiskIo.c

```
/*++

Copyright (c) 2004, Intel Corporation
All rights reserved. This program and the accompanying materials
are licensed and made available under the terms and conditions of the BSD
License
which accompanies this distribution.  The full text of the license may be
found at
http://opensource.org/licenses/bsd-license.php

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.

Module Name:

    DiskIo.c

Abstract:

  Disk IO protocol as defined in the EFI 1.0 specification.

  The Disk IO protocol is used to convert block oriented devices into
byte
  oriented devices. The Disk IO protocol is intended to layer on top of
the
  Block IO protocol.

--*/

#include "EfiSpec.h"
#include EFI_PROTOCOL_DEFINITION (DiskIo)

EFI_GUID  gEfiDiskIoProtocolGuid = EFI_DISK_IO_PROTOCOL_GUID;

EFI_GUID_STRING(&gEfiDiskIoProtocolGuid, "DiskIo Protocol", "EFI 1.0 Disk
IO Protocol");
```

D.2 EFI Global Variable GUID – GlobalVariable.h

```
/*++

Copyright (c) 2004, Intel Corporation
All rights reserved. This program and the accompanying materials
are licensed and made available under the terms and conditions of the BSD
License
which accompanies this distribution.  The full text of the license may be
found at
http://opensource.org/licenses/bsd-license.php

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.

Module Name:

    GlobalVariable.h

Abstract:

  GUID for EFI (NVRAM) Variables. Defined in EFI 1.0.

--*/
```

```
#ifndef _GLOBAL_VARIABLE_GUID_H_
#define _GLOBAL_VARIABLE_GUID_H_

#define EFI_GLOBAL_VARIABLE_GUID \
  { \
    0x8BE4DF61, 0x93CA, 0x11d2, 0xAA, 0x0D, 0x00, 0xE0, 0x98, 0x03, 0x2B,
0x8C \
  }

extern EFI_GUID gEfiGlobalVariableGuid;

#endif
```

D.3 EFI Global Variable GUID – GlobalVariable.c

```
/*++

Copyright (c) 2004, Intel Corporation
All rights reserved. This program and the accompanying materials
are licensed and made available under the terms and conditions of the BSD
License
which accompanies this distribution.  The full text of the license may be
found at
http://opensource.org/licenses/bsd-license.php

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.

Module Name:

    GlobalVariable.c

Abstract:

  GUID for EFI (NVRAM) Variables. Defined in EFI 1.0.

--*/

#include "EfiSpec.h"
#include EFI_GUID_DEFINITION (GlobalVariable)

EFI_GUID  gEfiGlobalVariableGuid = EFI_GLOBAL_VARIABLE_GUID;

EFI_GUID_STRING(&gEfiGlobalVariableGuid, "Efi", "Efi Variable GUID")
```

D.4 Disk I/O Driver - DiskIo.h

```
/*++

Copyright (c) 2004, Intel Corporation
All rights reserved. This program and the accompanying materials
are licensed and made available under the terms and conditions of the BSD
License
which accompanies this distribution.  The full text of the license may be
found at
http://opensource.org/licenses/bsd-license.php

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.

Module Name:

  DiskIo.h
```

```
Abstract:
  Private Data definition for Disk IO driver

--*/

#ifndef _DISK_IO_H
#define _DISK_IO_H

#include "Tiano.h"
#include "EfiDriverLib.h"

//
// Driver Consumed Protocol Prototypes
//
#include EFI_PROTOCOL_DEFINITION (DevicePath)
#include EFI_PROTOCOL_DEFINITION (BlockIo)

//
// Driver Produced Protocol Prototypes
//
#include EFI_PROTOCOL_DEFINITION (DriverBinding)
#include EFI_PROTOCOL_DEFINITION (ComponentName)
#include EFI_PROTOCOL_DEFINITION (DiskIo)

#define DISK_IO_PRIVATE_DATA_SIGNATURE  EFI_SIGNATURE_32 ('d', 's', 'k',
'I')

#define DATA_BUFFER_BLOCK_NUM           (64)

typedef struct {
  UINTN                 Signature;
  EFI_DISK_IO_PROTOCOL  DiskIo;
  EFI_BLOCK_IO_PROTOCOL *BlockIo;
} DISK_IO_PRIVATE_DATA;

#define DISK_IO_PRIVATE_DATA_FROM_THIS(a) CR (a, DISK_IO_PRIVATE_DATA,
DiskIo, DISK_IO_PRIVATE_DATA_SIGNATURE)

//
// Global Variables
//
extern EFI_DRIVER_BINDING_PROTOCOL  gDiskIoDriverBinding;
extern EFI_COMPONENT_NAME_PROTOCOL  gDiskIoComponentName;

#endif
```

D.5 Disk I/O Driver - DiskIo.c

```
/*++

Copyright (c) 2004 - 2005, Intel Corporation
All rights reserved. This program and the accompanying materials
are licensed and made available under the terms and conditions of the BSD
License
which accompanies this distribution.  The full text of the license may be
found at
http://opensource.org/licenses/bsd-license.php

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.

Module Name:

  DiskIo.c
```

```
Abstract:

  DiskIo driver that layers it's self on every Block IO protocol in the
system.
  DiskIo converts a block oriented device to a byte oriented device.

  ReadDisk may have to do reads that are not aligned on sector boundaries.
  There are three cases:

    UnderRun - The first byte is not on a sector boundary or the read
request is
               less than a sector in length.

    Aligned  - A read of N contiguous sectors.

    OverRun  - The last byte is not on a sector boundary.

--*/

#include "DiskIo.h"

//
// Prototypes
// Driver model protocol interface
//
EFI_STATUS
EFIAPI
DiskIoDriverEntryPoint (
  IN EFI_HANDLE           ImageHandle,
  IN EFI_SYSTEM_TABLE     *SystemTable
  );

EFI_STATUS
EFIAPI
DiskIoDriverBindingSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL    *This,
  IN EFI_HANDLE                     ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL       *RemainingDevicePath
  );

EFI_STATUS
EFIAPI
DiskIoDriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL    *This,
  IN EFI_HANDLE                     ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL       *RemainingDevicePath
  );

EFI_STATUS
EFIAPI
DiskIoDriverBindingStop (
  IN  EFI_DRIVER_BINDING_PROTOCOL    *This,
  IN  EFI_HANDLE                     ControllerHandle,
  IN  UINTN                          NumberOfChildren,
  IN  EFI_HANDLE                     *ChildHandleBuffer
  );

//
// Disk I/O Protocol Interface
//
EFI_STATUS
EFIAPI
DiskIoReadDisk (
  IN EFI_DISK_IO_PROTOCOL  *This,
  IN UINT32                MediaId,
  IN UINT64                Offset,
  IN UINTN                 BufferSize,
```

```
  OUT VOID                  *Buffer
  );

EFI_STATUS
EFIAPI
DiskIoWriteDisk (
  IN EFI_DISK_IO_PROTOCOL  *This,
  IN UINT32                MediaId,
  IN UINT64                Offset,
  IN UINTN                 BufferSize,
  IN VOID                  *Buffer
  );

EFI_DRIVER_BINDING_PROTOCOL gDiskIoDriverBinding = {
  DiskIoDriverBindingSupported,
  DiskIoDriverBindingStart,
  DiskIoDriverBindingStop,
  0x10,
  NULL,
  NULL
};

DISK_IO_PRIVATE_DATA        gDiskIoPrivateDataTemplate = {
  DISK_IO_PRIVATE_DATA_SIGNATURE,
  {
    EFI_DISK_IO_PROTOCOL_REVISION,
    DiskIoReadDisk,
    DiskIoWriteDisk
  },
  NULL
};

EFI_DRIVER_ENTRY_POINT (DiskIoDriverEntryPoint)

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

Routine Description:
  Register Driver Binding protocol for this driver.

Arguments:
  (Standard EFI Image entry - EFI_IMAGE_ENTRY_POINT)

Returns:
  EFI_SUCCESS - Driver loaded.
  other       - Driver not loaded.

--*/
{
  return EfiLibInstallAllDriverProtocols (
          ImageHandle,
          SystemTable,
          &gDiskIoDriverBinding,
          ImageHandle,
          &gDiskIoComponentName,
          NULL,
          NULL
          );
}

EFI_STATUS
EFIAPI
DiskIoDriverBindingSupported (
```

```
  IN EFI_DRIVER_BINDING_PROTOCOL  * This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     * RemainingDevicePath OPTIONAL
  )
/*++

  Routine Description:
    Test to see if this driver supports ControllerHandle. Any
ControllerHandle
    than contains a BlockIo protocol can be supported.

  Arguments:
    This                - Protocol instance pointer.
    ControllerHandle    - Handle of device to test.
    RemainingDevicePath - Not used.

  Returns:
    EFI_SUCCESS         - This driver supports this device.
    EFI_ALREADY_STARTED - This driver is already running on this device.
    other               - This driver does not support this device.

--*/
{
  EFI_STATUS            Status;
  EFI_BLOCK_IO_PROTOCOL *BlockIo;

  //
  // Open the IO Abstraction(s) needed to perform the supported test.
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiBlockIoProtocolGuid,
                (VOID **) &BlockIo,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  //
  // Close the I/O Abstraction(s) used to perform the supported test.
  //
  gBS->CloseProtocol (
        ControllerHandle,
        &gEfiBlockIoProtocolGuid,
        This->DriverBindingHandle,
        ControllerHandle
        );

  return EFI_SUCCESS;
}

EFI_STATUS
EFIAPI
DiskIoDriverBindingStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  * This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     * RemainingDevicePath OPTIONAL
  )
/*++

  Routine Description:
    Start this driver on ControllerHandle by opening a Block IO protocol
and
    installing a Disk IO protocol on ControllerHandle.

  Arguments:
```

```
    This                - Protocol instance pointer.
    ControllerHandle    - Handle of device to bind driver to.
    RemainingDevicePath - Not used, always produce all possible children.

  Returns:
    EFI_SUCCESS         - This driver is added to ControllerHandle.
    EFI_ALREADY_STARTED - This driver is already running on
ControllerHandle.
    other               - This driver does not support this device.

--*/
{
  EFI_STATUS            Status;
  DISK_IO_PRIVATE_DATA  *Private;

  Private = NULL;

  //
  // Connect to the Block IO interface on ControllerHandle.
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiBlockIoProtocolGuid,
                (VOID **) &gDiskIoPrivateDataTemplate.BlockIo,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );
  if (EFI_ERROR (Status)) {
    return Status;
  }
  //
  // Initialize the Disk IO device instance.
  //
  Private = EfiLibAllocateCopyPool (sizeof (DISK_IO_PRIVATE_DATA),
&gDiskIoPrivateDataTemplate);
  if (Private == NULL) {
    Status = EFI_OUT_OF_RESOURCES;
    goto ErrorExit;
  }
  //
  // Install protocol interfaces for the Disk IO device.
  //
  Status = gBS->InstallProtocolInterface (
                &ControllerHandle,
                &gEfiDiskIoProtocolGuid,
                EFI_NATIVE_INTERFACE,
                &Private->DiskIo
                );

ErrorExit:
  if (EFI_ERROR (Status)) {

    if (Private != NULL) {
      gBS->FreePool (Private);
    }

    gBS->CloseProtocol (
          ControllerHandle,
          &gEfiBlockIoProtocolGuid,
          This->DriverBindingHandle,
          ControllerHandle
          );
  }

  return Status;
}
```

```
EFI_STATUS
EFIAPI
DiskIoDriverBindingStop (
  IN  EFI_DRIVER_BINDING_PROTOCOL    *This,
  IN  EFI_HANDLE                     ControllerHandle,
  IN  UINTN                          NumberOfChildren,
  IN  EFI_HANDLE                     *ChildHandleBuffer
  )
/*++

  Routine Description:
    Stop this driver on ControllerHandle by removing Disk IO protocol and
closing
    the Block IO protocol on ControllerHandle.

  Arguments:
    This              - Protocol instance pointer.
    ControllerHandle  - Handle of device to stop driver on.
    NumberOfChildren  - Not used.
    ChildHandleBuffer - Not used.

  Returns:
    EFI_SUCCESS         - This driver is removed ControllerHandle.
    other               - This driver was not removed from this device.
    EFI_UNSUPPORTED

--*/
{
  EFI_STATUS            Status;
  EFI_DISK_IO_PROTOCOL  *DiskIo;
  DISK_IO_PRIVATE_DATA  *Private;

  //
  // Get our context back.
  //
  Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiDiskIoProtocolGuid,
                (VOID **) &DiskIo,
                This->DriverBindingHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_GET_PROTOCOL
                );
  if (EFI_ERROR (Status)) {
    return EFI_UNSUPPORTED;
  }

  Private = DISK_IO_PRIVATE_DATA_FROM_THIS (DiskIo);

  Status = gBS->UninstallProtocolInterface (
                ControllerHandle,
                &gEfiDiskIoProtocolGuid,
                &Private->DiskIo
                );
  if (!EFI_ERROR (Status)) {

    Status = gBS->CloseProtocol (
                  ControllerHandle,
                  &gEfiBlockIoProtocolGuid,
                  This->DriverBindingHandle,
                  ControllerHandle
                  );
  }

  if (!EFI_ERROR (Status)) {
    gBS->FreePool (Private);
  }
```

```
  return Status;
}

EFI_STATUS
EFIAPI
DiskIoReadDisk (
  IN EFI_DISK_IO_PROTOCOL  *This,
  IN UINT32                MediaId,
  IN UINT64                Offset,
  IN UINTN                 BufferSize,
  OUT VOID                 *Buffer
  )
/*++

  Routine Description:
    Read BufferSize bytes from Offset into Buffer.

    Reads may support reads that are not aligned on
    sector boundaries. There are three cases:

      UnderRun - The first byte is not on a sector boundary or the read
request is
                 less than a sector in length.

      Aligned  - A read of N contiguous sectors.

      OverRun  - The last byte is not on a sector boundary.


  Arguments:
    This       - Protocol instance pointer.
    MediaId    - Id of the media, changes every time the media is
replaced.
    Offset     - The starting byte offset to read from.
    BufferSize - Size of Buffer.
    Buffer     - Buffer containing read data.

  Returns:
    EFI_SUCCESS           - The data was read correctly from the device.
    EFI_DEVICE_ERROR      - The device reported an error while performing
the read.
    EFI_NO_MEDIA          - There is no media in the device.
    EFI_MEDIA_CHNAGED     - The MediaId does not matched the current
device.
    EFI_INVALID_PARAMETER - The read request contains device addresses
that are not
                            valid for the device.
    EFI_OUT_OF_RESOURCES

--*/
{
  EFI_STATUS             Status;
  DISK_IO_PRIVATE_DATA   *Private;
  EFI_BLOCK_IO_PROTOCOL *BlockIo;
  EFI_BLOCK_IO_MEDIA     *Media;
  UINT32                 BlockSize;
  UINT64                 Lba;
  UINT64                 OverRunLba;
  UINTN                  UnderRun;
  UINTN                  OverRun;
  BOOLEAN                TransactionComplete;
  UINTN                  WorkingBufferSize;
  UINT8                  *WorkingBuffer;
  UINTN                  Length;
  UINT8                  *Data;
  UINT8                  *PreData;
  UINTN                  IsBufferAligned;
  UINTN                  DataBufferSize;
```

```
  BOOLEAN               LastRead;

  Private   = DISK_IO_PRIVATE_DATA_FROM_THIS (This);

  BlockIo   = Private->BlockIo;
  Media     = BlockIo->Media;
  BlockSize = Media->BlockSize;

  if (Media->MediaId != MediaId) {
    return EFI_MEDIA_CHANGED;
  }

  WorkingBuffer     = Buffer;
  WorkingBufferSize = BufferSize;

  //
  // Allocate a temporary buffer for operation
  //
  DataBufferSize = BlockSize * DATA_BUFFER_BLOCK_NUM;

  if (Media->IoAlign > 1) {
    PreData = EfiLibAllocatePool (DataBufferSize + Media->IoAlign);
    Data    = PreData - ((UINTN) PreData & (Media->IoAlign - 1)) + Media-
>IoAlign;
  } else {
    PreData = EfiLibAllocatePool (DataBufferSize);
    Data    = PreData;
  }

  if (PreData == NULL) {
    return EFI_OUT_OF_RESOURCES;
  }

  Lba                 = DivU64x32 (Offset, BlockSize, &UnderRun);

  Length              = BlockSize - UnderRun;
  TransactionComplete = FALSE;

  Status              = EFI_SUCCESS;
  if (UnderRun != 0) {
    //
    // Offset starts in the middle of an Lba, so read the entire block.
    //
    Status = BlockIo->ReadBlocks (
                        BlockIo,
                        MediaId,
                        Lba,
                        BlockSize,
                        Data
                        );

    if (EFI_ERROR (Status)) {
      goto Done;
    }

    if (Length > BufferSize) {
      Length              = BufferSize;
      TransactionComplete = TRUE;
    }

    EfiCopyMem (WorkingBuffer, Data + UnderRun, Length);

    WorkingBuffer += Length;

    WorkingBufferSize -= Length;
    if (WorkingBufferSize == 0) {
      goto Done;
    }
```

```
    Lba += 1;
  }

  OverRunLba = Lba + DivU64x32 (WorkingBufferSize, BlockSize, &OverRun);

  if (!TransactionComplete && WorkingBufferSize >= BlockSize) {
    //
    // If the DiskIo maps directly to a BlockIo device do the read.
    //
    if (OverRun != 0) {
      WorkingBufferSize -= OverRun;
    }
    //
    // Check buffer alignment
    //
    IsBufferAligned = (UINTN) WorkingBuffer & (UINTN) (Media->IoAlign -
1);

    if (Media->IoAlign <= 1 || IsBufferAligned == 0) {
      //
      // Alignment is satisfied, so read them together
      //
      Status = BlockIo->ReadBlocks (
                          BlockIo,
                          MediaId,
                          Lba,
                          WorkingBufferSize,
                          WorkingBuffer
                          );

      if (EFI_ERROR (Status)) {
        goto Done;
      }

      WorkingBuffer += WorkingBufferSize;

    } else {
      //
      // Use the allocated buffer instead of the original buffer
      // to avoid alignment issue.
      // Here, the allocated buffer (8-byte align) can satisfy the
alignment
      //
      LastRead = FALSE;
      do {
        if (WorkingBufferSize <= DataBufferSize) {
          //
          // It is the last calling to readblocks in this loop
          //
          DataBufferSize  = WorkingBufferSize;
          LastRead        = TRUE;
        }

        Status = BlockIo->ReadBlocks (
                            BlockIo,
                            MediaId,
                            Lba,
                            DataBufferSize,
                            Data
                            );
        if (EFI_ERROR (Status)) {
          goto Done;
        }

        EfiCopyMem (WorkingBuffer, Data, DataBufferSize);
        WorkingBufferSize -= DataBufferSize;
        WorkingBuffer += DataBufferSize;
```

```
        Lba += DATA_BUFFER_BLOCK_NUM;
      } while (!LastRead);
    }
  }

  if (!TransactionComplete && OverRun != 0) {
    //
    // Last read is not a complete block.
    //
    Status = BlockIo->ReadBlocks (
                        BlockIo,
                        MediaId,
                        OverRunLba,
                        BlockSize,
                        Data
                        );

    if (EFI_ERROR (Status)) {
      goto Done;
    }

    EfiCopyMem (WorkingBuffer, Data, OverRun);
  }

Done:
  if (PreData != NULL) {
    gBS->FreePool (PreData);
  }

  return Status;
}

EFI_STATUS
EFIAPI
DiskIoWriteDisk (
  IN EFI_DISK_IO_PROTOCOL  *This,
  IN UINT32                MediaId,
  IN UINT64                Offset,
  IN UINTN                 BufferSize,
  IN VOID                  *Buffer
  )
/*++

  Routine Description:
    Read BufferSize bytes from Offset into Buffer.

    Writes may require a read modify write to support writes that are not
    aligned on sector boundaries. There are three cases:

      UnderRun - The first byte is not on a sector boundary or the write
request
                 is less than a sector in length. Read modify write is
required.

      Aligned  - A write of N contiguous sectors.

      OverRun  - The last byte is not on a sector boundary. Read modified
write
                 required.

  Arguments:
    This       - Protocol instance pointer.
    MediaId    - Id of the media, changes every time the media is
replaced.
    Offset     - The starting byte offset to read from.
    BufferSize - Size of Buffer.
    Buffer     - Buffer containing read data.
```

```
  Returns:
    EFI_SUCCESS           - The data was written correctly to the device.
    EFI_WRITE_PROTECTED   - The device can not be written to.
    EFI_DEVICE_ERROR      - The device reported an error while performing
the write.
    EFI_NO_MEDIA          - There is no media in the device.
    EFI_MEDIA_CHNAGED     - The MediaId does not matched the current
device.
    EFI_INVALID_PARAMETER - The write request contains device addresses
that are not
                            valid for the device.
    EFI_OUT_OF_RESOURCES

--*/
{
  EFI_STATUS            Status;
  DISK_IO_PRIVATE_DATA  *Private;
  EFI_BLOCK_IO_PROTOCOL *BlockIo;
  EFI_BLOCK_IO_MEDIA    *Media;
  UINT32                BlockSize;
  UINT64                Lba;
  UINT64                OverRunLba;
  UINTN                 UnderRun;
  UINTN                 OverRun;
  BOOLEAN               TransactionComplete;
  UINTN                 WorkingBufferSize;
  UINT8                 *WorkingBuffer;
  UINTN                 Length;
  UINT8                 *Data;
  UINT8                 *PreData;
  UINTN                 IsBufferAligned;
  UINTN                 DataBufferSize;
  BOOLEAN               LastWrite;

  Private   = DISK_IO_PRIVATE_DATA_FROM_THIS (This);

  BlockIo   = Private->BlockIo;
  Media     = BlockIo->Media;
  BlockSize = Media->BlockSize;

  if (Media->ReadOnly) {
    return EFI_WRITE_PROTECTED;
  }

  if (Media->MediaId != MediaId) {
    return EFI_MEDIA_CHANGED;
  }

  DataBufferSize = BlockSize * DATA_BUFFER_BLOCK_NUM;

  if (Media->IoAlign > 1) {
    PreData = EfiLibAllocatePool (DataBufferSize + Media->IoAlign);
    Data    = PreData - ((UINTN) PreData & (Media->IoAlign - 1)) + Media-
>IoAlign;
  } else {
    PreData = EfiLibAllocatePool (DataBufferSize);
    Data    = PreData;
  }

  if (PreData == NULL) {
    return EFI_OUT_OF_RESOURCES;
  }

  WorkingBuffer       = Buffer;
  WorkingBufferSize   = BufferSize;

  Lba                 = DivU64x32 (Offset, BlockSize, &UnderRun);
```

```
  Length              = BlockSize - UnderRun;
  TransactionComplete = FALSE;

  Status              = EFI_SUCCESS;
  if (UnderRun != 0) {
    //
    // Offset starts in the middle of an Lba, so do read modify write.
    //
    Status = BlockIo->ReadBlocks (
                        BlockIo,
                        MediaId,
                        Lba,
                        BlockSize,
                        Data
                        );

    if (EFI_ERROR (Status)) {
      goto Done;
    }

    if (Length > BufferSize) {
      Length              = BufferSize;
      TransactionComplete = TRUE;
    }

    EfiCopyMem (Data + UnderRun, WorkingBuffer, Length);

    Status = BlockIo->WriteBlocks (
                        BlockIo,
                        MediaId,
                        Lba,
                        BlockSize,
                        Data
                        );
    if (EFI_ERROR (Status)) {
      goto Done;
    }

    WorkingBuffer += Length;
    WorkingBufferSize -= Length;
    if (WorkingBufferSize == 0) {
      goto Done;
    }

    Lba += 1;
  }

  OverRunLba = Lba + DivU64x32 (WorkingBufferSize, BlockSize, &OverRun);

  if (!TransactionComplete && WorkingBufferSize >= BlockSize) {
    //
    // If the DiskIo maps directly to a BlockIo device do the write.
    //
    if (OverRun != 0) {
      WorkingBufferSize -= OverRun;
    }
    //
    // Check buffer alignment
    //
    IsBufferAligned = (UINTN) WorkingBuffer & (UINTN) (Media->IoAlign -
1);

    if (Media->IoAlign <= 1 || IsBufferAligned == 0) {
      //
      // Alignment is satisfied, so write them together
      //
      Status = BlockIo->WriteBlocks (
                          BlockIo,
```

```
                        MediaId,
                        Lba,
                        WorkingBufferSize,
                        WorkingBuffer
                        );

      if (EFI_ERROR (Status)) {
        goto Done;
      }

      WorkingBuffer += WorkingBufferSize;

    } else {
      //
      // The buffer parameter is not aligned with the request
      // So use the allocated instead.
      // It can fit almost all the cases.
      //
      LastWrite = FALSE;
      do {
        if (WorkingBufferSize <= DataBufferSize) {
          //
          // It is the last calling to writeblocks in this loop
          //
          DataBufferSize  = WorkingBufferSize;
          LastWrite       = TRUE;
        }

        EfiCopyMem (Data, WorkingBuffer, DataBufferSize);
        Status = BlockIo->WriteBlocks (
                          BlockIo,
                          MediaId,
                          Lba,
                          DataBufferSize,
                          Data
                          );
        if (EFI_ERROR (Status)) {
          goto Done;
        }

        WorkingBufferSize -= DataBufferSize;
        WorkingBuffer += DataBufferSize;
        Lba += DATA_BUFFER_BLOCK_NUM;
      } while (!LastWrite);
    }
  }

  if (!TransactionComplete && OverRun != 0) {
    //
    // Last bit is not a complete block, so do a read modify write.
    //
    Status = BlockIo->ReadBlocks (
                        BlockIo,
                        MediaId,
                        OverRunLba,
                        BlockSize,
                        Data
                        );

    if (EFI_ERROR (Status)) {
      goto Done;
    }

    EfiCopyMem (Data, WorkingBuffer, OverRun);

    Status = BlockIo->WriteBlocks (
                        BlockIo,
                        MediaId,
```

```
                        OverRunLba,
                        BlockSize,
                        Data
                        );
    if (EFI_ERROR (Status)) {
      goto Done;
    }
  }

Done:
  if (PreData != NULL) {
    gBS->FreePool (PreData);
  }

  return Status;
}
```

D.6 Disk I/O Driver - ComponentName.c

```
/*++

Copyright (c) 2004, Intel Corporation
All rights reserved. This program and the accompanying materials
are licensed and made available under the terms and conditions of the BSD
License
which accompanies this distribution.  The full text of the license may be
found at
http://opensource.org/licenses/bsd-license.php

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED.

Module Name:

  ComponentName.c

Abstract:

--*/

#include "DiskIo.h"

//
// EFI Component Name Functions
//
EFI_STATUS
EFIAPI
DiskIoComponentNameGetDriverName (
  IN  EFI_COMPONENT_NAME_PROTOCOL  *This,
  IN  CHAR8                        *Language,
  OUT CHAR16                       **DriverName
  );

EFI_STATUS
EFIAPI
DiskIoComponentNameGetControllerName (
  IN  EFI_COMPONENT_NAME_PROTOCOL                     *This,
  IN  EFI_HANDLE                                      ControllerHandle,
  IN  EFI_HANDLE                                      ChildHandle
OPTIONAL,
  IN  CHAR8                                           *Language,
  OUT CHAR16                                          **ControllerName
  );

//
```

```
// EFI Component Name Protocol
//
EFI_COMPONENT_NAME_PROTOCOL     gDiskIoComponentName = {
  DiskIoComponentNameGetDriverName,
  DiskIoComponentNameGetControllerName,
  "eng"
};

static EFI_UNICODE_STRING_TABLE mDiskIoDriverNameTable[] = {
  {
    "eng",
    L"Generic Disk I/O Driver"
  },
  {
    NULL,
    NULL
  }
};

EFI_STATUS
EFIAPI
DiskIoComponentNameGetDriverName (
  IN  EFI_COMPONENT_NAME_PROTOCOL  *This,
  IN  CHAR8                        *Language,
  OUT CHAR16                       **DriverName
  )
/*++

  Routine Description:
    Retrieves a Unicode string that is the user readable name of the EFI
Driver.

  Arguments:
    This       - A pointer to the EFI_COMPONENT_NAME_PROTOCOL instance.
    Language   - A pointer to a three character ISO 639-2 language
identifier.
                 This is the language of the driver name that that the
caller
                 is requesting, and it must match one of the languages
specified
                 in SupportedLanguages.  The number of languages
supported by a
                 driver is up to the driver writer.
    DriverName - A pointer to the Unicode string to return.  This Unicode
string
                 is the name of the driver specified by This in the
language
                 specified by Language.

  Returns:
    EFI_SUCCESS           - The Unicode string for the Driver specified
by This
                            and the language specified by Language was
returned
                            in DriverName.
    EFI_INVALID_PARAMETER - Language is NULL.
    EFI_INVALID_PARAMETER - DriverName is NULL.
    EFI_UNSUPPORTED       - The driver specified by This does not support
the
                            language specified by Language.

--*/
{
  return EfiLibLookupUnicodeString (
          Language,
          gDiskIoComponentName.SupportedLanguages,
          mDiskIoDriverNameTable,
          DriverName
```

```
          );
}

EFI_STATUS
EFIAPI
DiskIoComponentNameGetControllerName (
  IN  EFI_COMPONENT_NAME_PROTOCOL                     *This,
  IN  EFI_HANDLE                                      ControllerHandle,
  IN  EFI_HANDLE                                      ChildHandle
OPTIONAL,
  IN  CHAR8                                           *Language,
  OUT CHAR16                                          **ControllerName
  )
/*++

  Routine Description:
    Retrieves a Unicode string that is the user readable name of the
controller
    that is being managed by an EFI Driver.

  Arguments:
    This             - A pointer to the EFI_COMPONENT_NAME_PROTOCOL
instance.
    ControllerHandle - The handle of a controller that the driver
specified by
                       This is managing.  This handle specifies the
controller
                       whose name is to be returned.
    ChildHandle      - The handle of the child controller to retrieve the
name
                       of.  This is an optional parameter that may be
NULL.  It
                       will be NULL for device drivers.  It will also be
NULL
                       for a bus drivers that wish to retrieve the name
of the
                       bus controller.  It will not be NULL for a bus
driver
                       that wishes to retrieve the name of a child
controller.
    Language         - A pointer to a three character ISO 639-2 language
                       identifier.  This is the language of the
controller name
                       that that the caller is requesting, and it must
match one
                       of the languages specified in SupportedLanguages.
The
                       number of languages supported by a driver is up to
the
                       driver writer.
    ControllerName   - A pointer to the Unicode string to return.  This
Unicode
                       string is the name of the controller specified by
                       ControllerHandle and ChildHandle in the language
specified
                       by Language from the point of view of the driver
specified
                       by This.

  Returns:
    EFI_SUCCESS           - The Unicode string for the user readable name
in the
                            language specified by Language for the driver
                            specified by This was returned in DriverName.
    EFI_INVALID_PARAMETER - ControllerHandle is not a valid EFI_HANDLE.
    EFI_INVALID_PARAMETER - ChildHandle is not NULL and it is not a valid
EFI_HANDLE.
    EFI_INVALID_PARAMETER - Language is NULL.
```

```
    EFI_INVALID_PARAMETER - ControllerName is NULL.
    EFI_UNSUPPORTED       - The driver specified by This is not currently
managing
                            the controller specified by ControllerHandle
and
                            ChildHandle.
    EFI_UNSUPPORTED       - The driver specified by This does not support
the
                            language specified by Language.

--*/
{
  return EFI_UNSUPPORTED;
}
```

Appendix E
EDK Sample Drivers

This appendix lists sample peripheral drivers that are available in the *EDK.* Table 65 defines the column headers used in the following table (Table 66).

Table 65. Column descriptions

Field	Field Value	Description
IA32		Y if the driver is included in the default build of this tip
IA64		Y if the driver is included in the default build of this tip
EBC		Y if the driver is included in the default build of this tip
DUET		Y if the driver is included in the default build of this tip
X64		Y if the driver is included in the default build of this tip
DB		Number of Driver Binding Protocols installed in the driver entry point.
CFG		Y if the Driver Configuration Protocol is installed in the driver entry point.
DIAG		Y if the Driver Diagnostics Protocol is installed in the driver entry point.
CN		Y if the Component Name Protocol is installed in the driver entry point.
Class	B	Bus driver.
	D	Device driver.
	H	Hybrid driver.
	RB	Root bridge driver.
	S	Service driver.
	I	Initializing driver.
Child	All	All child handles in first call to Start().
	1/All	Can create 1 child handle at a time or all child handles in Start().
	1	Creates at most 1 child handle in Start().
	0	Create no child handles in Start(). Used for hot-plug bus types.
Parent		Number of parent drivers to this driver
BS or RT	BS	EFI Boot Services driver.
	RT	EFI Runtime Services driver.
UL		Y if the driver is unloadable.
HP		Y if the driver supports a Hot Plug device or bus

Table 66. EDK sample peripheral driver properties

Driver	IA32	IA64	EBC	DUET	X64	DB	CFG	DIAG	CN	CLASS	Child	Parent	BS or RT	UL	HP
Sample\Bus\Isa\Isabus			Y	Y		1			Y	B	All	1	BS		
Sample\Bus\Isa\IsaFloppy			Y	Y		1			Y	D		1	BS		
Sample\Bus\Isa\IsaSerial			Y	Y		1			Y	B	1	1	BS		
Sample\Bus\Isa\Ps2Keyboard			Y	Y		1			Y	D		1	BS		
Sample\Bus\Isa\Ps2Mouse			Y	Y		1			Y	D		1	BS		
Sample\Bus\Pci\AtapiExtPassThru		Y	Y			1			Y	D		1	BS	Y	
Sample\Bus\Pci\AtapiPassThru		Y	Y			1			Y	D		1	BS	Y	
Sample\Bus\Pci\Ehci			Y	Y		1			Y	D		1	BS		
Sample\Bus\Pci\EhciRouting			Y			1			Y	D		1	BS		
Sample\Bus\Pci\IdeBus	Y	Y	Y	Y	Y	1	Y	Y	Y	B	All	1	BS		
Sample\Bus\Pci\IdeController				Y		1			Y	D		1	BS		
Sample\Bus\Pci\PciBus	Y	Y				1			Y	B	1/All	1	BS		Y
Sample\Bus\Pci\PciBusNoEnumeration				Y	Y	1			Y	B	1/All	1	BS		
Sample\Bus\Pci\Uhci	Y	Y	Y	Y	Y	1			Y	D		1	BS		
Sample\Bus\Pci\Undi		Y	Y	Y	Y	1				B	1	1	RT		

Sample\Bus\Pci\VgaMiniPort			Y	Y		1			Y	D		1	BS		
Sample\Bus\Scsi\ScsiBus	Y	Y	Y		Y	1			Y	B	1/All	1	BS		
Sample\Bus\Scsi\ScsiDisk	Y	Y	Y		Y	1			Y	D		1	BS		
Sample\Bus\Usb\UsbBus	Y		Y	Y	Y	1			Y	B	0	1	BS		
Sample\Bus\Usb\UsbKb	Y	Y	Y	Y	Y	1			Y	D		1	BS		
Sample\Bus\Usb\UsbMassStorage	Y	Y	Y	Y	Y	1			Y	D		1	BS		
Sample\Bus\Usb\UsbMouse	Y	Y	Y	Y	Y	1			Y	D		1	BS		
Sample\Bus\WinNtThunk\BlockIo	Y					1	Y	Y	Y	D		1	BS		
Sample\Bus\WinNtThunk\Console	Y					1			Y	D		1	BS		
Sample\Bus\WinNtThunk\Gop	Y					1			Y	B	1	1	BS		
Sample\Bus\WinNtThunk\SerialIo	Y					1			Y	B	1	1	BS		
Sample\Bus\WinNtThunk\SimpleFileSystem	Y					1			Y	D		1	BS		
Sample\Bus\WinNtThunk\WinNtBusDriver	Y					1			Y	H	1/All	1	BS		

Appendix F Glossary

Table 67 defines terms that are used in this document. See the glossary in the *UEFI 2.0 Specification* for definitions of additional terms.

Table 67. Definitions of Terms

Term	Definition
<Enumerated Type>	Element of an enumeration. Type INTN.
ACPI	Advanced Configuration and Power Interface.
ANSI	American National Standards Institute.
API	Application programming interface.
ASCII	American Standard Code for Information Interchange.
ATAPI	Advanced Technology Attachment Packet Interface.
BAR	Base Address Register.
BBS	BIOS Boot Specification.
BC	Base Code.
BEV	Bootstrap Entry Vector. A pointer that points to code inside an option ROM that will directly load an OS.
BIOS	Basic input/output system.
BIS	Boot Integrity Services.
BM	Boot manager.
BOOLEAN	Logical Boolean. 1-byte value containing a 0 for **FALSE** or a 1 for **TRUE**. Other values are undefined.
BOT	Bulk-Only Transport.
BS	EFI Boot Services Table or EFI Boot Service(s).
CBI	Control/Bulk/Interrupt Transport.
CBW	Command Block Wrapper.
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.
CHAR8	1-byte character.
CID	Compatible ID.
CONST	Declares a variable to be of type const. This modifier is a hint to the compiler to enable optimization and stronger type checking at compile time.

CR	Containing Record.
CRC	Cyclic Redundancy Check.
CSW	Command Status Wrapper.
DAC	Dual Address Cycle.
DHCP4	Dynamic Host Configuration Protocol Version 4.
DID	Device ID.
DIG64	Developer's Interface Guide for 64-bit Intel Architecture-based Servers.
DMA	Direct Memory Access.
EBC	EFI Byte Code.
ECR	Engineering Change Request.
EFI	Extensible Firmware Interface.
EFI_EVENT	Handle to an event structure. Type VOID *.
EFI_GUID	128-bit buffer containing a unique identifier value. Unless otherwise specified, aligned on a 64-bit boundary.
EFI_HANDLE	A collection of related interfaces. Type VOID *.
EFI_IP_ADDRESS	16-byte buffer aligned on a 4-byte boundary. An IPv4 or IPv6 internet protocol address.
EFI_Ipv4_ADDRESS	4-byte buffer. An IPv4 internet protocol address.
EFI_Ipv6_ADDRESS	16-byte buffer. An IPv6 internet protocol address.
EFI_LBA	Logical block address. Type UINT64.
EFI_MAC_ADDRESS	32-byte buffer containing a network Media Access Controller address.
EFI_STATUS	Status code. Type INTN.
EFI_TPL	Task priority level. Type UINTN.
EISA	Extended Industry Standard Architecture.
FAT	File allocation table.
FIFO	First In First Out.
FPSWA	Floating Point Software Assist.
FRU	Field Replaceable Unit.
FTP	File Transfer Protocol.
GPT	Guided Partition Table.
GUID	Globally Unique Identifier.
HC	Host controller.
HID	Hardware ID.
I/O	Input/output.
IA-32	32-bit Intel architecture.
IBV	Independent BIOS vendor.
IDE	Integrated Drive Electronics.

IEC	International Electrotechnical Commission.
IHV	Independent hardware vendor.
IN	Datum is passed to the function.
INT	Interrupt.
INT16	2-byte signed value.
INT32	4-byte signed value.
INT64	8-byte signed value.
INT8	1-byte signed value.
INTN	Signed value of native width. (4 bytes on IA-32, 8 bytes on Itanium architecture operations)
IPF	Itanium processor family.
Ipv4	Internet Protocol Version 4.
Ipv6	Internet Protocol Version 6.
ISA	Industry Standard Architecture.
ISO	Industry Standards Organization.
iSCSI	SCSI protocol over TCP/IP.
KB	Keyboard.
LAN	Local area network.
LUN	Logical Unit Number.
MAC	Media Access Controller.
MMIO	Memory Mapped I/O.
NIC	Network interface controller.
NII	Network Interface Identifier.
NVRAM	Nonvolatile RAM.
OEM	Original equipment manufacturer.
OHCI	Open Host Controller Interface.
OpROM	Option ROM.
OPTIONAL	Datum that is passed to the function is optional, and a **NULL** may be passed if the value is not supplied.
OS	Operating system.
OUT	Datum is returned from the function.
PCI	Peripheral Component Interconnect.
PCMCIA	Personal Computer Memory Card International Association.
PE	Portable Executable.
PE/COFF	PE32, PE32+, or Common Object File Format.
PNPID	Plug and Play ID.
POST	Power On Self Test.

PPP	Point-to-Point Protocol.
PUN	Physical Unit Number.
PXE	Preboot Execution Environment.
PXE BC (or PxeBc)	PXE Base Code Protocol.
QH	Queue Head.
RAID	Redundant Array of Inexpensive Disks.
RAM	Random access memory.
ROM	Read-only memory.
RT	EFI Runtime Table and EFI Runtime Service(s).
SAL	System Abstraction Layer.
SCSI	Small Computer System Interface.
SIG	Special Interest Group.
S.M.A.R.T.	Self-Monitoring Analysis Reporting Technology.
SMBIOS	System Management BIOS.
SMBus	System Management Bus.
SNP	Simple Network Protocol.
SPT	SCSI Pass Thru.
ST	EFI System Table
STATIC	The function has local scope. This modifier replaces the standard C static key word, so it can be overloaded for debugging.
TCP/IP	Transmission Control Protocol/Internet Protocol.
TD	Transfer Descriptor.
TPL	Task Priority Level.
UART	Universal Asynchronous Receiver-Transmitter.
UHCI	Universal Host Controller Interface.
UID	Unique ID.
UINT16	2-byte unsigned value.
UINT32	4-byte unsigned value.
UINT64	8-byte unsigned value.
UINT8	1-byte unsigned value.