

Driver Writer's Guide for UEFI 2.3.1

03/08/2012

Version 1.01

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Except for a limited copyright license to copy this specification for internal use only, no license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

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

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

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

Intel, the Intel logo and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other names and brands may be claimed as the property of others.

Copyright © 2003–2012 Intel Corporation.

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. Updated the coding conventions. 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 sections 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
0.97	Clarify role of EDK as being implementation-specific and added definitions for the myriad of library references so the meaning of the implementation specific code examples could be clarified without having to reference documents aside from the <i>UEFI Specification</i> .	6/25/2008
0.98	Updated for UEFI 2.3.1 and EDK II	2/12/12
1.00	Review feedback incorporated, additional formatting	2/27/12
1.01	Review feedback incorporated	3/8/12

Contents

Revision History.....	ii	
Contents	iii	
1	Introduction.....	1
1.1	Overview.....	1
1.1.1	Assumptions	1
1.2	Organization of this document	2
1.3	Related information	3
1.3.1	UEFI Specifications	3
1.3.2	Industry Standard Bus Specifications	3
1.3.3	Other specifications	4
1.3.4	EDK II and UDK2010 Development Kit	5
1.4	Typographic conventions.....	5
2	UEFI Driver Implementation Checklist	7
2.1	Design and implementation of UEFI drivers	9
2.2	How to implement features in EDK II	12
3	Foundation.....	15
3.1	Basic programming model.....	15
3.2	Objects managed by UEFI-based firmware	16
3.3	UEFI system table	17
3.4	Handle database	18
3.5	GUIDs.....	22
3.6	Protocols and handles	23
3.6.1	Protocols are produced and consumed	24
3.6.2	Protocol interface structure.....	25
3.6.3	Protocols provided in addition to the <i>UEFI Specification</i>	26
3.6.4	Multiple protocol instances.....	27
3.6.5	Tag GUID	27
3.7	UEFI images	27
3.7.1	Applications	31
3.7.2	Drivers.....	31
3.8	Events and task priority levels	32
3.8.1	Defining priority	35
3.8.2	Creating locks	35
3.8.3	Using callbacks	35
3.9	UEFI device paths	36
3.9.1	How drivers use device paths.....	39
3.9.2	IPF Considerations for device path data structures.....	40
3.9.3	Environment variables	40
3.10	UEFI driver model	41
3.10.1	Device driver	42
3.10.2	Bus driver	44
3.10.3	Hybrid driver	46

3.11	Service Drivers	46
3.12	Root Bridge Driver.....	46
3.13	Initializing Driver.....	47
3.14	UEFI Driver Model Connection Process	47
3.14.1	ConnectController()	48
3.14.2	Loading UEFI option ROM drivers.....	51
3.14.3	DisconnectController()	51
3.15	Platform initialization	51
3.15.1	Connecting PCI Root Bridges.....	53
3.15.2	Connecting the PCI bus	54
3.15.3	Connecting consoles	55
3.15.4	Console drivers	56
3.15.5	Console variables	59
3.15.6	ConIn	59
3.15.7	ConOut	62
3.15.8	ErrOut	64
3.15.9	Boot Manager Connect All Processing.....	64
3.15.10	Boot Manager Driver List Processing	65
3.15.11	Boot Manager BootNext Processing.....	65
3.15.12	Boot Manager Boot Option Processing	66
4	General Driver Design Guidelines.....	67
4.1	Common Coding Practices	67
4.1.1	Type Checking	68
4.1.2	Avoid Name Collisions.....	69
4.1.3	Maximize Warning Levels	69
4.1.4	Compiler Optimizations	69
4.2	Maximize Platform Compatibility.....	69
4.2.1	Never Assume all UEFI Drivers are Executed.....	69
4.2.2	Eliminate System Memory Assumptions	70
4.2.3	Use UEFI Memory Allocation Services.....	70
4.2.4	Do not make assumptions about I/O subsystem configurations	71
4.2.5	Never Directly Access Hardware Resources.....	71
4.2.6	Memory ordering.....	72
4.2.7	DMA	73
4.2.8	Supporting Older EFI Specifications and UEFI Specifications.....	73
4.2.9	Reduce Poll Frequency	74
4.2.10	Minimize Time in Notification Functions	74
4.2.11	Use Proper Task Priority Levels	74
4.2.12	Design to be re-entrant.....	75
4.2.13	Do not use hidden PCI Option ROM Regions.....	75
4.2.14	Store Configuration Data with Device.....	76
4.2.15	Do not use hard-coded device path nodes	76
4.2.16	Do not cause errors on shared storage devices	77
4.2.17	Limit use of Console Services.....	77
4.2.18	Offer alternatives to function keys	78
4.3	Maximize CPU Compatibility.....	79
4.3.1	Assignment and comparison operators.....	80
4.3.2	Casting pointers	84
4.3.3	Converting pointers	84

4.3.4	UEFI Data Type Sizes.....	86
4.3.5	Negative Numbers.....	86
4.3.6	Returning Pointers in a Function Parameter	87
4.3.7	Array Subscripts.....	88
4.3.8	Piecemeal Structure Allocations.....	89
4.4	Optimization Techniques	89
4.4.1	Size Reduction	90
4.4.2	Performance Optimizations	91
4.4.3	CopyMem() and SetMem() Operations	92
5	UEFI Services.....	95
5.1	Services that UEFI drivers commonly use	97
5.1.1	Memory Allocation Services	98
5.1.2	Miscellaneous Services	104
5.1.3	Handle Database and Protocol Services	107
5.1.4	Task Priority Level(TPL) Services	128
5.1.5	Event services.....	130
5.1.6	SetTimer()	139
5.1.7	Stall()	143
5.2	Services that UEFI drivers rarely use	145
5.2.1	ConnectController() and DisconnectController().....	147
5.2.2	ReinstallProtocolInterface()	150
5.2.3	LocateDevicePath().....	152
5.2.4	LoadImage() and StartImage()	154
5.2.5	GetVariable() and SetVariable()	156
5.2.6	QueryVariableInfo ()	160
5.2.7	GetTime()	160
5.2.8	CalculateCrc32()	161
5.2.9	ConvertPointer()	163
5.2.10	InstallConfigurationTable()	166
5.2.11	GetNextMonotonicCount()	170
5.3	Services that UEFI drivers should not use.....	170
5.3.1	InstallProtocolInterface().....	172
5.3.2	UninstallProtocolInterface()	172
5.3.3	HandleProtocol()	173
5.3.4	LocateHandle()	173
5.3.5	ProtocolsPerHandle()	173
5.3.6	RegisterProtocolNotify()	173
5.3.7	UnloadImage().....	173
5.3.8	GetNextVariableName()	174
5.3.9	SetWatchdogTimer()	176
5.3.10	SetTime(), GetWakeupTime(), and SetWakeupTime()	176
5.3.11	GetMemoryMap()	176
5.3.12	ExitBootServices()	176
5.3.13	SetVirtualAddressMap().....	176
5.3.14	QueryCapsuleCapabilities()	177
5.3.15	UpdateCapsule()	177
5.3.16	ResetSystem()	177
5.3.17	Exit()	177
5.3.18	GetNextHighMonotonicCount()	178

6	UEFI Driver Categories	179
6.1	Device drivers.....	179
6.1.1	Required Device Driver Features	179
6.1.2	Optional Device Driver Features	180
6.1.3	Compatibility with Older EFI/UEFI Specifications.....	181
6.1.4	Device drivers with one driver binding protocol	181
6.1.5	Device drivers with multiple driver binding protocols...	183
6.1.6	Device driver protocol management.....	185
6.2	Bus drivers	188
6.2.1	Required Bus Driver Features.....	188
6.2.2	Optional Bus Driver Features	189
6.2.3	Bus drivers with one driver binding protocol	190
6.2.4	Bus drivers with multiple driver binding protocols.....	190
6.2.5	Bus driver protocol and child management	191
6.2.6	Bus drivers that produce one child in Start()	192
6.2.7	Bus drivers that produce all children in Start()	193
6.2.8	Bus drivers that produce at most one child in Start()	193
6.2.9	Bus drivers that produce no children in Start()	194
6.2.10	Bus drivers that produce children with multiple parents.....	194
6.3	Hybrid drivers.....	194
6.3.1	Required Hybrid Driver Features.....	195
6.3.2	Optional Hybrid Driver Features	195
6.4	Service Drivers	197
6.5	Root Bridge Drivers	198
6.6	Initializing Drivers	198
7	Driver Entry Point	199
7.1	Optional Features	201
7.2	UEFI Driver Model	202
7.2.1	Single Driver Binding Protocol	204
7.2.2	Multiple Driver Binding Protocols	208
7.2.3	Adding Driver Health Protocol Feature.....	209
7.2.4	Adding Driver Family Override Protocol Feature.....	211
7.3	Adding the Driver Supported EFI Version Protocol Feature.....	212
7.4	Adding HII Packages Feature	214
7.5	Adding HII Config Access Protocol Feature.....	216
7.6	Adding the Unload Feature	217
7.7	Adding the Exit Boot Services feature	222
7.8	Initializing Driver entry point	226
7.9	Service Driver entry point	227
7.10	Root bridge driver entry point	229
7.11	Runtime Drivers.....	233
8	Private Context Data Structures	241
8.1	Containing Record Macro.....	241
8.2	Data structure design	243
8.3	Allocating private context data structures.....	246
8.4	Freeing private context data structures	249
8.5	Retrieving private context data structures	251

9	Driver Binding Protocol.....	253
	9.1 Driver Binding Protocol Implementations.....	253
	9.2 Driver Binding Protocol Template	255
	9.3 Testing Driver Binding Protocol	256
10	UEFI Service Binding Protocol.....	257
	10.1 Service Binding Protocol Implementations	258
	10.2 Service Driver.....	258
	10.3 UEFI Driver Model Driver	260
11	UEFI Driver and Controller Names.....	261
	11.1 Component Name Protocol Implementations.....	262
	11.2 GetDriverName() Implementations.....	265
	11.3 GetControllerName() Implementations.....	266
	11.3.1 Device Drivers	267
	11.3.2 Bus Drivers and Hybrid Drivers	271
	11.4 Testing Component Name Protocols.....	273
12	UEFI Driver Configuration	275
	12.1 HII overview.....	275
	12.1.1 HII Database and Package Lists	276
	12.2 General steps for implementing HII functionality.....	277
	12.3 HII Protocols	279
	12.3.1 HII Database Protocol and HII String Protocol.....	279
	12.3.2 HII Config Routing Protocol	283
	12.3.3 HII Config Access Protocol.....	284
	12.3.4 Rarely used HII protocols	289
	12.4 HII functionality	289
	12.4.1 Branding, and displaying a banner.....	289
	12.4.2 Specifying supported languages	290
	12.4.3 Specifying configuration information.....	290
	12.4.4 Making configuration data available to other drivers	291
	12.4.5 Check to see if configuration parameters are valid.....	292
	12.5 Forms and VFR files.....	292
	12.6 HII Implementation Recommendations	294
	12.6.1 Minimize callbacks.....	294
	12.6.2 Don't reparse the package list.....	295
	12.6.3 Concentrate on critical aspects of the driver	295
	12.6.4 Perform usability testing	296
	12.7 Porting to UEFI HII functionality.....	296
13	UEFI Driver Diagnostics	297
	13.1 Driver Diagnostics Protocol Implementations	298
	13.2 RunDiagnostics() Implementations	300
	13.2.1 Device Drivers	301
	13.2.2 Bus Drivers and Hybrid Drivers	303
	13.2.3 RunDiagnostics() as a UEFI Application	306
	13.3 Testing Driver Diagnostics Protocols	306
14	Driver Health Protocol	309

14.1	Driver Health Protocol Implementation.....	309
14.2	GetHealthStatus() Implementations.....	311
14.2.1	Device Drivers	313
14.2.2	Bus Drivers and Hybrid Drivers	315
14.3	Repair() Implementation.....	316
14.3.1	Device Drivers	317
14.3.2	Bus Drivers and Hybrid Drivers	318
15	Driver Family Override Protocol	321
15.1	Driver Family Override Protocol Implementation.....	321
15.2	GetVersion() Implementation.....	323
16	Driver Supported EFI Version Protocol	325
16.1	Driver Supported EFI Version Protocol Implementation	325
17	Bus-Specific Driver Override Protocol	329
17.1	Bus Specific Driver Override Protocol Implementation.....	329
17.2	Private Context Data Structure	330
17.3	Bus Specific Driver Override Protocol Installation	331
17.4	GetDriver() Implementation	332
17.5	Adding Driver Image Handles	333
18	PCI Driver Design Guidelines.....	335
18.1	PCI Root Bridge I/O Protocol Drivers.....	336
18.2	PCI Bus Drivers.....	336
18.2.1	Hot-plug PCI buses.....	337
18.3	PCI drivers	337
18.3.1	Supported()	338
18.3.2	Start() and Stop()	341
18.3.3	PCI Cards with Multiple PCI Controllers	346
18.4	Accessing PCI resources.....	347
18.4.1	Memory-mapped I/O ordering issues	348
18.4.2	Hardfail/Softfail.....	349
18.4.3	When a PCI device does not receive resources	352
18.5	PCI DMA	353
18.5.1	Map() Service Cautions	353
18.5.2	Weakly ordered memory transactions	354
18.5.3	Bus Master Read and Write Operations.....	354
18.5.4	Bus Master Common Buffer Operations	356
18.5.5	4 GB Memory Boundary	356
18.5.6	DMA Bus Master Read Operation	357
18.5.7	DMA Bus Master Write Operation.....	359
18.5.8	DMA Bus Master Common Buffer Operation	362
18.6	PCI Optimization Techniques	364
18.6.1	PCI I/O fill operations	365
18.6.2	PCI I/O FIFO operations	366
18.6.3	PCI I/O CopyMem() Operations	367
18.6.4	PCI Configuration Header Operations	368
18.6.5	PCI I/O MMIO Buffer Operations.....	370
18.6.6	PCI I/O Polling Operations	370

18.7	PCI Option ROM Images.....	372
18.7.1	EfiRom Utility.....	372
18.7.2	Using INF File to Generate PCI Option ROM Image	374
18.7.3	Using FDF File to Generate PCI Option ROM Image	375
19	USB Driver Design Guidelines.....	377
19.1	USB Host Controller Driver	380
19.1.1	Driver Binding Protocol Supported().....	380
19.1.2	Driver Binding Protocol Start()	382
19.1.3	Driver Binding Protocol Stop()	383
19.1.4	USB 2 Host Controller Protocol Data Transfer Services	383
19.2	USB Bus Driver	386
19.3	USB Device Driver.....	386
19.3.1	Driver Binding Protocol Supported().....	387
19.3.2	Driver Binding Protocol Start() and Stop()	388
19.3.3	I/O Protocol Implementations	390
19.3.4	State machine consideration	392
19.4	Debug Techniques	393
19.4.1	Debug Message Output	393
19.4.2	USB Bus Analyzer.....	393
19.4.3	USBCheck/USBCV Tool.....	394
19.5	Nonconforming USB Devices.....	394
20	SCSI Driver Design Guidelines.....	395
20.1	SCSI Host Controller Driver	395
20.1.1	Single-Channel SCSI Adapters	396
20.1.2	Multi-Channel SCSI Adapters	397
20.1.3	SCSI Adapters with RAID	398
20.1.4	Implementing driver binding protocol.....	400
20.1.5	Implementing Extended SCSI Pass Thru Protocol	401
20.1.6	SCSI command set device considerations	404
20.1.7	Discover a SCSI channel	407
20.1.8	SCSI Device Path	407
20.1.9	Using Extended SCSI Pass Thru Protocol	408
20.2	SCSI Bus Driver	410
20.3	SCSI Device Driver.....	410
20.3.1	Driver Binding Protocol Supported().....	410
20.3.2	Driver Binding Protocol Start() and Stop()	412
20.3.3	I/O Protocol Implementations	412
21	ATA Driver Design Guidelines	413
21.1	ATA Host Controller Driver.....	413
21.1.1	Implementing Driver Binding Protocol	414
21.1.2	Implementing ATA Pass Thru Protocol	415
21.1.3	ATA Command Set Considerations	417
21.1.4	ATA Device Paths	417
21.6	ATA Bus Driver	418
22	Text Console Driver Design Guidelines.....	419
22.1	Assumptions.....	420
22.2	Simple Text Input Protocol Implementation	420

22.2.1	Reset() Implementation	421
22.2.2	ReadKeyStroke() and ReadKeyStrokeEx() Implementation	421
22.2.3	WaitForKey and WaitForKeyEx Notification Implementation	422
22.2.4	SetState() Implementation.....	422
22.2.5	RegisterKeyNotify() Implementation.....	422
22.2.6	UnregisterKeyNotify() Implementation.....	422
22.3	Simple Text Output Protocol Implementation.....	422
22.3.1	Reset() Implementation	423
22.3.2	OutputString() Implementation.....	424
22.3.3	TestString() Implementation.....	424
22.3.4	QueryMode() Implementation	424
22.3.5	SetMode() Implementation.....	425
22.3.6	SetAttribute() Implementation	425
22.3.7	ClearScreen() Implementation	426
22.3.8	SetCursorPosition() Implementation	426
22.3.9	EnableCursor() Implementation	426
22.4	Serial I/O Protocol Implementations	427
22.4.1	Reset() Implementation	427
22.4.2	SetAttributes() Implementation.....	428
22.4.3	SetControl() and GetControl() Implementation.....	428
22.4.4	Write() and Read() Implementation.....	429
23	Graphics Driver Design Guidelines	431
23.1	Assumptions.....	431
23.2	Graphics Output Protocol Implementation	432
23.2.1	Single output graphics adapters	433
23.2.2	Multiple output graphics adapters.....	433
23.2.3	Driver Binding Protocol Implementation	434
23.2.4	QueryMode(), SetMode(), and Blt() Implementation	435
23.3	EDID Discovered Protocol Implementation.....	438
23.4	EDID Active Protocol Implementation	438
23.5	EDID Override Protocol Implementation	439
23.5.1	GetEdid() Implementation	440
24	Mass Storage Driver Design Guidelines.....	441
24.1	Assumptions.....	442
24.2	Block I/O Protocol Implementations.....	442
24.2.1	Reset() Implementation	443
24.2.2	ReadBlocks() and ReadBlocksEx() Implementation	444
24.2.3	WriteBlocks() and WriteBlockEx() Implementation.....	444
24.2.4	FlushBlocks() and FlushBlocksEx() Implementation.....	445
24.3	Storage Security Protocol Implementation.....	445
25	Network Driver Design Guidelines	447
25.4	Assumptions.....	449
25.5	NII Protocol and UNDI Implementations.....	449
25.5.1	Exit Boot Services Event.....	452
25.5.2	Set Virtual Address Map Event	453
25.5.3	Memory leaks caused by UNDI	453
25.6	Simple Network Protocol Implementations.....	453

26	25.7 Managed Network Protocol Implementations.....	455
	User Credential Driver Design Guidelines	457
	26.1 Assumptions.....	457
	26.2 User Credential Protocol Implementation.....	457
27	Load File Driver Design Guidelines	459
	27.1 Assumptions.....	459
	27.2 Load File Protocol Implementation.....	459
	27.2.1 LoadFile() Implementation.....	460
28	IPF Platform Porting Considerations	461
	28.1 General notes about porting to IPF platforms.....	461
	28.2 Alignment Faults	462
	28.3 Casting Pointers.....	462
	28.4 Packed Structures	464
	28.5 UEFI Device Paths	465
	28.6 PCI Configuration Header 64-bit BAR	466
	28.7 Speculation and floating point register usage.....	468
29	EFI Byte Code Porting Considerations	469
	29.1 No Assembly Support	469
	29.2 No C++ Support	469
	29.3 No Floating Point Support.....	469
	29.4 Use of sizeof()	470
	29.4.1 Global Variable Initialization	471
	29.4.2 CASE Statements	472
	29.5 Natural Integers and Fixed Size Integers.....	472
	29.6 Memory ordering.....	473
	29.7 Performance considerations	473
	29.7.1 Performance considerations for data types	473
	29.8 UEFI Driver Entry Point	474
30	Building UEFI Drivers	475
	30.1 Prerequisites.....	475
	30.2 Create EDK II Package.....	476
	30.3 Create UEFI Driver Directory.....	477
	30.3.1 Disk I/O Driver Example.....	479
	30.3.2 Reserved Directory Names	479
	30.3.3 EBC Virtual Machine Driver Example	480
	30.4 Adding a UEFI Driver to DSC File.....	480
	30.5 Building a UEFI driver	481
31	Testing and Debugging UEFI Drivers	483
	31.1 Native and EBC	483
	31.2 Compiler Optimizations	483
	31.3 UEFI Shell Debugging	484
	31.3.1 Testing Specific Protocols	484
	31.3.2 Other Testing.....	485
	31.3.3 Loading UEFI drivers.....	486

31.3.4	Unloading UEFI drivers.....	487
31.3.5	Connecting UEFI Drivers.....	487
31.3.6	Driver and Device Information	489
31.3.7	Testing the Driver Configuration Protocol.....	492
31.3.8	Testing the Driver Diagnostics Protocols.....	493
31.4	Debugging code statements	493
31.4.1	Configuring DebugLib with EDK II.....	495
31.4.2	Capturing Debug Messages.....	497
31.5	POST codes	497
31.5.1	POST Card Debug.....	497
31.5.2	Other options.....	499
32	Distributing UEFI Drivers	501
32.1	PCI Option ROM	501
32.2	Integrated in Platform FLASH.....	501
32.3	EFI System Partition	501
Appendix A EDK II File Templates		503
A.1	UEFI Driver Template	505
A.1.1	<<DriverName>>.inf File for a UEFI Driver	506
A.1.2	<<DriverName>>.inf File for a UEFI Runtime Driver	507
A.1.3	<<DriverName>>.h File	509
A.1.4	<<DriverName>>.c File.....	512
A.1.5	<<ProtocolName>>.c File	514
A.2	UEFI Driver Optional Protocol Templates	515
A.2.1	ComponentName.c File	515
A.2.2	DriverConfiguration.c File	516
A.2.3	HiiConfigAccess.c File.....	517
A.2.4	DriverHealth.c File.....	518
A.2.5	DriverFamilyOverride.c File.....	519
A.2.6	BusSpecificDriverOverride.c File	520
A.2.7	DriverDiagnostics.c File.....	520
A.3	UEFI Driver I/O Protocol Templates	521
A.3.1	Usb2Hc.c File	521
A.3.2	ExtScsiPassThru.c File.....	525
A.3.3	AtaPassThru.c File	526
A.3.4	SimpleTextInput.c File	528
A.3.5	SimpleTextOutput.c File	529
A.3.6	SerialIo.c File.....	531
A.3.7	GraphicsOutput.c File.....	533
A.3.8	BlockIo.c File	534
A.3.9	NiiUndi.c File.....	536
A.3.10	SimpleNetwork.c File	537
A.3.11	UserCredential.c File	540
A.3.12	LoadFile.c File	542
A.4	Platform Specific UEFI Driver Templates.....	542
A.4.1	EdidOverride.c File	542
A.5	EDK II Package Extension Templates	543
A.5.1	Protocol File Template.....	543
A.5.2	GUID File Template	545
A.5.3	Library Class File Template.....	546

A.5.4 Including Protocols, GUIDs, and Library Classes	547
Appendix B EDK II Sample Drivers.....	549
Appendix C Glossary	552

Figures

Figure 1—Object managed by UEFI-based firmware	17
Figure 2—Handle database	19
Figure 3—Handle types.....	20
Figure 4—Construction of a protocol.....	26
Figure 5—Image types	29
Figure 6—Event types	33
Figure 7—Booting sequence for UEFI operational model.....	52
Figure 8—Sample system configuration	53
Figure 9—Device driver with single Driver Binding Protocol	182
Figure 10—Device driver with optional features	183
Figure 11—Device driver with multiple Driver Binding Protocols	184
Figure 12—Device driver protocol management	186
Figure 13—Complex device driver protocol management	187
Figure 14—Bus driver protocol management	192
Figure 15—Testing Component Name Protocol GetDriverName().....	274
Figure 16—Testing Component Name Protocol GetControllerName()	274
Figure 17—Testing Driver Diagnostics Protocols.....	307
Figure 18—Driver Health Status State Diagram	313
Figure 19—PCI driver stack.....	336
Figure 20—A multi-controller PCI device	346
Figure 21—USB driver stack.....	378
Figure 22—Sample SCSI driver stack on single-channel adapter	397
Figure 23—Sample SCSI driver implementation on a multichannel adapter	398
Figure 24—Sample SCSI driver implementation on multichannel RAID adapter.....	399
Figure 25—Text Console geometry.....	426
Figure 26—Example single-output graphics driver Implementation.....	433
Figure 27—Example dual-output graphics driver implementation	434
Figure 28—BIOS buffer	437
Figure 29—UEFI UNDI Network Stack.....	450
Figure 30—SNP-based network stack	454

Tables

Table 1—Organization of the <i>UEFI Driver Writer's Guide</i>	2
Table 2—Classes of UEFI drivers to develop	10
Table 3—Protocols produced by various devices.....	11
Table 4—Mapping operations to UEFI drivers.....	12
Table 5—Description of handle types.....	20

Table 6—Description of image types.....	29
Table 7—Description of event types	33
Table 8—Task priority levels defined in UEFI	34
Table 9—Types of device path nodes defined in <i>UEFI Specification</i>	37
Table 10—Protocols separating the loading and starting/stopping of drivers	41
Table 11—I/O protocols produced in the Start() function for different device classes	43
Table 12—Connecting controllers: Driver connection precedence rules	48
Table 13—UEFI console drivers.....	56
Table 14—Alternate key sequences for remote terminals	79
Table 15—Space optimizations	90
Table 16—Speed optimizations	91
Table 17—Alphabetical listing of UEFI services.....	95
Table 18—UEFI services that are commonly used by UEFI drivers	97
Table 19—UEFI services that are rarely used by UEFI drivers	146
Table 20—UEFI services that should not be used by UEFI drivers.....	171
Table 21—UEFI Driver Feature Selection Matrix	201
Table 22—Service Binding Protocols.....	257
Table 23—Health Status Values	312
Table 24—UEFI Specific Revision Values	327
Table 25—Classes of PCI drivers.....	335
Table 26—PCI Attributes.....	344
Table 27—EDK II attributes #defines	345
Table 28—PCI BAR attributes	345
Table 29—PCI Embedded Device Attributes.....	346
Table 30—Classes of USB drivers.....	377
Table 31—Classes of SCSI drivers.....	395
Table 32—SCSI device path examples	408
Table 33—Classes of ATA drivers	413
Table 34—SATA device path examples	417
Table 35—Serial I/O protocol control bits.....	428
Table 36—Network driver differences	447
Table 37—!PXE interface structure.....	452
Table 38—CDB structure.....	452
Table 39—Reserved directory names.....	479
Table 40—UEFI Shell commands.....	484
Table 41—Other Shell Testing Procedures	485
Table 42—UEFI Shell commands for loading UEFI drivers.....	486
Table 43—UEFI Shell commands for unloading UEFI drivers	487
Table 44—UEFI Shell commands for connecting UEFI drivers.....	487
Table 45—UEFI Shell commands for driver and device information.....	490
Table 46—Error levels	494
Table 47—UEFI Driver Properties	549
Table 48—Sample UEFI Driver Properties.....	550
Table 49—Definitions of terms.....	552

Examples

Example 1—EFI_GUID data structure in EDK II	23
Example 2—Protocol structure in EDK II	24
Example 3—Device Path Header	37
Example 4—PCI Device Path	37
Example 5—Device Path Examples.....	39
Example 6—ConnectController() UEFI Boot Service.....	48
Example 7—Stronger type checking	68
Example 8—Assignment operation warnings.....	82
Example 9—Comparison operation warnings	83
Example 10—Examples of casting pointers.....	85
Example 11—Negative number example.....	87
Example 12—Casting OUT function parameters	88
Example 13—Array subscripts example	89
Example 14—Incorrect and correct piecemeal structure allocation	89
Example 15—CopyMem() and SetMem() Speed Optimizations	93
Example 16—Allocate and free pool using UEFI Boot Services Table	100
Example 17—Allocate and free pool using MemoryAllocationLib	101
Example 18—Allocate and clear pool using MemoryAllocationLib	101
Example 19—Allocate and initialize pool using MemoryAllocationLib	102
Example 20—Allocate and free pages using UEFI Boot Services Table.....	103
Example 21—Allocate and free pages using MemoryAllocationLib.....	103
Example 22—Allocate and free aligned pages using MemoryAllocationLib.....	104
Example 23—Allocate and clear a buffer using UEFI Boot Services	105
Example 24—Allocate and clear a buffer using BaseMemoryLib.....	105
Example 25—Allocate and clear a buffer using BaseMemoryLib.....	106
Example 26—Allocate and copy buffer	106
Example 27—Allocate and clear a buffer using BaseMemoryLib.....	107
Example 28—Install protocols in UEFI Driver entry point.....	110
Example 29—Install protocols in UEFI Driver entry point using UefiLib.....	112
Example 30—Uninstall protocols in UEFI Driver Unload() function.	112
Example 31—Add child handle to handle database	113
Example 32—Remove child handle from handle database.	114
Example 33—Add tag GUID to a controller handle.....	114
Example 34—Remove tag GUID from a controller handle.	115
Example 35—Retrieve all handles in handle database.....	116
Example 36—Retrieve all Block I/O Protocols in handle database	117
Example 37—Locate first Decompress Protocol in handle database.....	118
Example 38—OpenProtocol() function prototype	120
Example 39—OpenProtocol() TEST_PROTOCOL.....	122
Example 40—OpenProtocol() GET_PROTOCOL	122
Example 41—OpenProtocol() EFI_OPEN_PROTOCOL_BY_DRIVER.....	124
Example 42—OpenProtocol() EFI_OPEN_PROTOCOL_BY_DRIVER EFI_OPEN_PROTOCOL_EXCLUSIVE	125
Example 43—OpenProtocol() EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER	126

Example 44—Count child handles using OpenProtocolInformation()	128
Example 45—Using TPL Services for a Global Lock	129
Example 46—Using UEFI Library for a Global Lock	130
Example 47—Create and close a wait event	133
Example 48—Create and Close an Exit Boot Services Event	134
Example 49—Create and Close an Exit Boot Services Event Group	135
Example 50—Create and Signal an Event Group	136
Example 51—Signal a key press event	137
Example 52—Wait for one-shot timer event to be signaled	139
Example 53—Create periodic timer event	141
Example 54—Create one-shot timer event	142
Example 55—Cancel and close one-shot timer event	143
Example 56—Fixed delay stall	144
Example 57—Poll for completion status using stalls	145
Example 58—Recursive connect in response to a hot-add operation	148
Example 59—Recursive disconnect in response to a hot-remove operation	148
Example 60—Disconnect a UEFI Driver from all handles	149
Example 61—Reinstall Block I/O Protocol for media change	151
Example 62—Reinstall Device Path Protocol for Serial I/O attributes change	152
Example 63—Locate Device Path	154
Example 64—Load and Start a UEFI Application from a PCI Option ROM	156
Example 65—Write configuration structure to a UEFI variable	158
Example 66—Read configuration structure from a UEFI variable	159
Example 67—Use UefiLib to read configuration structure from a UEFI variable	159
Example 68—Collect information about the UEFI variable store	160
Example 69—Get time and date	161
Example 70—Get real time clock capabilities	161
Example 71—Calculate and update 32-bit CRC in UEFI System Table	161
Example 72—Calculate and 32-bit CRC for a structure	162
Example 73—Verify 32-bit CRC in UEFI System Table	162
Example 74—Create a Set Virtual Address Map event	164
Example 75—Convert a global pointer from physical to virtual	164
Example 76—Using UefiRuntimeLib to convert a pointer	165
Example 77—Using UefiRuntimeLib to convert a function pointer	166
Example 78—Using UefiRuntimeLib to convert a linked list	166
Example 79—Add or update a configuration table entry	168
Example 80—Add or update a configuration table entry	168
Example 81—Wait for key press or timer event	170
Example 82—Retrieve 64-bit monotonic counter value	170
Example 83—Print all UEFI variable store contents	175
Example 84—ResetSystem	177
Example 85—Exit from a UEFI Driver	178
Example 86—UEFI Driver Entry Point	200
Example 87—UEFI Driver INF File	200
Example 88—EDK II UefiLib driver initialization functions	203
Example 89—Single Driver Binding Protocol	206
Example 90—Single Driver Binding Protocol with optional features	207

Example 91—Multiple Driver Binding Protocols	209
Example 92—Driver Heath Protocol Feature	210
Example 93—Driver Family Override Protocol Feature	212
Example 94—Driver Supported EFI Version Protocol Feature	214
Example 95—HII Packages feature	215
Example 96—UEFI Driver INF File with HII Packages feature	216
Example 97—HII Config Access Protocol Feature	217
Example 98—Add the Unload feature	219
Example 99—UEFI Driver INF File with Unload feature.....	220
Example 100—UEFI Driver Model Unload Feature.....	221
Example 101—Adding the Exit Boot Services feature.....	224
Example 102—Add the Unload and Exit Boot Services event features	226
Example 103—Initializing driver entry point	227
Example 104—Service driver entry point using image handle	228
Example 105—Service driver entry point creating new handle	228
Example 106—Single PCI root bridge driver entry point	231
Example 107—Multiple PCI root bridge driver entry point.....	233
Example 108—UEFI Runtime Driver entry point	236
Example 109—UEFI Runtime Driver INF File.....	237
Example 110—UEFI Runtime Driver entry point with Unload feature	239
Example 111—Containing record macro definitions	243
Example 112—Containing record macro definitions	243
Example 113—Simple private context data structure.....	245
Example 114—Complex private context data structure	246
Example 115—Allocation of a private context data structure	247
Example 116—Library allocation of private context data structure.....	247
Example 117—Disk I/O allocation of private context data structure.....	248
Example 118—Free a private context data structure	250
Example 119—Disk I/O free of a private context data structure.....	250
Example 120—Retrieving the Disk I/O private context data structure	251
Example 121—Retrieving the disk I/O private context data structure in Stop()	252
Example 122—Driver Binding Protocol	254
Example 123—Driver Binding Protocol declaration	255
Example 124—Service Binding Protocol	258
Example 125—Service Binding Protocol for Service Driver.....	260
Example 126—Component Name Protocol.....	263
Example 127—Component Name 2 Protocol	263
Example 128—Driver Diagnostics Protocol declaration.....	264
Example 129—GetDriverName() for Device, Bus, or Hybrid Driver	266
Example 130—GetControllerName () Service.....	266
Example 131—GetControllerName() for a Device Driver	268
Example 132—Controller names in private context data structure	268
Example 133—Adding a controller name to a dynamic controller name table	269
Example 134—Freeing a dynamic controller name table.....	270
Example 135—Device driver with dynamic controller names.....	271
Example 136—GetControllerName() for a Bus Driver or Hybrid Driver	273
Example 137—Example of a Unicode string file.....	278

Example 138—Example of a Unicode string file.....	278
Example 139—ExtractConfig() Function	287
Example 140—RouteConfig() Function.....	288
Example 141—Callback function	288
Example 142—Unicode string file with support for multiple languages	290
Example 143—Sample VFR file, simplified.....	294
Example 144—Driver Diagnostics Protocol	299
Example 145—Driver Diagnostics 2 Protocol.....	299
Example 146—Driver Diagnostics Protocol declaration.....	299
Example 147—RunDiagnostics() Service	301
Example 148—RunDiagnostics() for a Device Driver.....	303
Example 149—RunDiagnostics() for a Bus Driver or Hybrid Driver	305
Example 150—Driver Health Protocol	310
Example 151—Install Driver Health Protocol.....	311
Example 152—GetHealthStatus() Function of the Driver Health Protocol.....	311
Example 153—GetHealthStatus() for a Device Driver	314
Example 154—GetHealthStatus() for a Bus Driver or Hybrid Driver	316
Example 155—Repair() Function for a Device Driver	318
Example 156—Repair() for a Bus Driver or Hybrid Driver	319
Example 157—Driver Family Override Protocol	322
Example 158—Install Driver Family Override Protocol.....	323
Example 159—GetVersion() Function of the Driver Family Override Protocol	323
Example 160—Driver Support EFI Version Protocol	326
Example 161—Driver Supported EFI Version Protocol Feature	326
Example 162—Bus Specific Driver Override Protocol.....	330
Example 163—Private Context Data Structure with a Bus Specific Driver Override Protocol	331
Example 164—Private Context Data Structure Initialization.....	331
Example 165—Install Bus Specific Driver Override Protocol.....	331
Example 166—Uninstall Bus Specific Driver Override Protocol	332
Example 167—GetDriver() Function of a Bus Specific Driver Override Protocol	333
Example 168—Adding Driver Image Handles.....	334
Example 169—Supported() Reading partial PCI Configuration Header.....	339
Example 170—Supported() Reading entire PCI Configuration Header	341
Example 171—Start() for a 64-bit DMA-capable PCI controller.....	343
Example 172—Restore PCI Attributes in Stop().....	343
Example 173—Completing a memory write transaction.....	349
Example 174—Accessing ISA resources on a PCI controller	350
Example 175—Locate PCI handles with matching bus number	352
Example 176—Map() Function	354
Example 177—Completing a bus master write operation.....	355
Example 178—Bus master read operation.....	359
Example 179—Bus master write operation	362
Example 180—Allocate bus master common buffer	364
Example 181—Free bus master common buffer	364
Example 182—PCI I/O 8-bit fill with a loop	365
Example 183—PCI I/O 32-bit fill with a loop.....	365

Example 184—PCI I/O 8-bit fill without a loop	366
Example 185—PCI I/O 32-bit fill without a loop	366
Example 186—PCI I/O FIFO using a loop	367
Example 187—PCI I/O FIFO without a loop	367
Example 188—Scroll frame buffer using a loop	368
Example 189—Scroll frame buffer without a loop	368
Example 190—Read PCI configuration using a loop	369
Example 191—Read PCI configuration 32 bits at a time	369
Example 192—Read PCI configuration 32 bits at a time	369
Example 193—Write 1MB Frame Buffer using a loop	370
Example 194—Write 1MB Frame Buffer with no loop	370
Example 195—Using Mem.Read() and Stall() to poll for 1 second	371
Example 196—Using PollIo() to poll for 1 second	371
Example 197—EfiRom Utility Help	373
Example 198—EfiRom Utility Dump Feature	374
Example 199—UEFI Driver INF File for PCI Option ROM	375
Example 200—Specify name of FDF file from a DSC file	376
Example 201—Using an FDF file to Generate PCI Option ROM images	376
Example 202—USB 2 Host Controller Protocol	379
Example 203—USB I/O Protocol	379
Example 204—Supported() service for USB host controller driver	382
Example 205—Disable USB Legacy Support	382
Example 206—Supported() for a USB device driver	388
Example 207—USB mass storage driver private context data structure	389
Example 208—USB Mouse Private Context Data Structure	390
Example 209—Setup asynchronous interrupt transfer for USB mouse driver	390
Example 210—Completing an asynchronous interrupt transfer	392
Example 211—Retrieving pointer movement	392
Example 212—Extended SCSI Pass Thru Protocol	401
Example 213—SCSI Pass Thru Mode Structure for Single Channel Adapter	402
Example 214—SCSI Pass Thru Mode Structure for Multi-Channel Adapter	402
Example 215—SCSI Pass Thru Mode Structures for RAID SCSI adapter	403
Example 216—Building Device Path for ATAPI Device	405
Example 217—Non-Blocking Extended SCSI Pass Thru Protocol Implementation	407
Example 218—Blocking and non-blocking modes	409
Example 219—Supported() for a SCSI device driver	411
Example 220—ATA Pass Thru Protocol	415
Example 221—ATA Pass Thru Mode Structure	416
Example 222—SCSI Pass Thru Mode Structures for RAID SCSI adapter	416
Example 223—Simple Text Input Protocol	421
Example 224—Simple Text Input Ex Protocol	421
Example 225—Simple Text Output Protocol	423
Example 226—Light reset of terminal driver	423
Example 227—Full reset of terminal driver	424
Example 228—Query current Simple Text Output Mode	425
Example 229—Query all Simple Text Output Modes	425

Example 230—Simple Text Output Protocol.....	427
Example 231—Graphics Output Protocol	433
Example 232—Graphics Output Protocol Blt() Service.....	436
Example 233—EDID Discovered Protocol	438
Example 234—EDID Active Protocol	439
Example 235—DID Override Protocol	439
Example 236—Block I/O Protocol.....	443
Example 237—Block I/O 2 Protocol	443
Example 238—Storage Security Command Protocol	446
Example 239—Network Interface Identifier Protocol.....	451
Example 240—Simple Network Protocol.....	455
Example 241—User Credential Protocol	458
Example 242—Load File Protocol	460
Example 243—Pointer-cast alignment fault	463
Example 244—Corrected pointer-cast alignment fault.....	463
Example 245—Packed structure alignment fault	464
Example 246—Corrected packed structure alignment fault	465
Example 247—UEFI device path node alignment fault	466
Example 248—Corrected UEFI device path node alignment fault	466
Example 249—Accessing a 64-bit BAR in a PCI configuration header.....	467
Example 250—Size of data types with EBC	470
Example 251—Global Variable Initialization that fails for EBC	471
Example 252—Global Variable Initialization that works for EBC.....	471
Example 253—Case statements that fail for EBC.....	472
Example 254—Case statements that work for EBC	472
Example 255—EDK II Package Directory.....	476
Example 256—EDK II Package DEC File	477
Example 257—EDK II Package DSC File.....	477
Example 258—UEFI Driver Directory	478
Example 259—UEFI Driver INF File	478
Example 260—UEFI Driver C Source File.....	479
Example 261—Disk I/O UEFI Driver Source Files.....	479
Example 262—EBC driver with instruction set architecture-specific files.....	480
Example 263—EDK II Package DSC File	481
Example 264—Build Output Directory.....	482
Example 265—EDK II Package DSC File with Optimizations Disabled.....	483
Example 266—EDK II Package DSC File with Build Options	495
Example 267—PcdDebugPropertyMask bitmask	496
Example 268—EDK II Package DSC File with Build Options	497
Example 269—UEFI Driver Entry Point with POST_CODE() Macros	498
Example 270—Enable POST_CODE() macros from DSC file	499
Example A-1—UEFI Driver INF file template	507
Example A-2—UEFI Runtime Driver INF file template	509
Example A-3—UEFI Driver include file template	511
Example A-4—UEFI Driver implementation template	513
Example A-5—UEFI Driver protocol implementation template	515
Example A-6—Component Name Protocol implementation template	516

Example A-7—Driver Configuration Protocol implementation template.....	517
Example A-8—Driver Health Protocol implementation template.....	518
Example A-9—Driver Health Protocol implementation template.....	519
Example A-10—Driver Family Override Protocol implementation template.....	519
Example A-11—Bus Specific Driver Override Protocol implementation template	520
Example A-12—Driver Diagnostics Protocols implementation template	521
Example A-13—USB 2 Host Controller Protocol implementation template.....	524
Example A-14—Extended SCSI Pass Thru Protocol implementation template.....	526
Example A-15—ATA Pass Thru Protocol implementation template	527
Example A-16—Simple Text Input Protocols implementation template.....	529
Example A-17—Simple Text Output Protocol implementation template	531
Example A-18—Serial I/O Protocol implementation template.....	532
Example A-19—Graphics Output Protocol implementation template.....	534
Example A-20—Block I/O, Block I/O 2, and Storage Security Protocols implementation template.....	536
Example A-21—Network Interface Identifier Protocol implementation template	537
Example A-22—Simple Network Protocol implementation template	539
Example A-23—User Credential Protocol implementation template.....	542
Example A-24—Load File Protocol implementation template.....	542
Example A-25—EDID Override Protocol implementation template	543
Example A-26—Add protocol to an EDK II package	543
Example A-27—Protocol include file template	544
Example A-28—Add GUID to an EDK II package	545
Example A-29—GUID include file template	545
Example A-30—Add Library Class to an EDK II package.....	546
Example A-31—Library Class include file template	547
Example A-32—Protocol, GUID, and Library Class include statements	548
Example A-33—Protocol and GUID INF statements	548

1 *Introduction*

1.1 Overview

UEFI is a modular, extensible interface that abstracts the details of platform hardware from an operating system (OS). It complements existing interfaces, helps manufacturers create OS-neutral add-in products, and provides an efficient replacement for PC BIOS legacy option ROMs.

This document is designed to aid in the development of UEFI Drivers using the EDK II open source project as a development environment. The EDK II provides a cross-platform firmware development environment for UEFI. UEFI Drivers are described in the *Unified Extensible Firmware Interface Specification*, Version 2.3.1 (hereafter referred to as the “*UEFI Specification*”). There are different categories of UEFI Drivers, and many variations of each category. This document provides basic information for the most common categories of UEFI drivers. Many other driver designs are possible.

In addition, this document covers the design guidelines and recommendations for the different driver-related UEFI Protocols, along with the design guidelines for PCI, USB, SCSI, ATA, Consoles, Serial Ports, Graphics, Mass Storage, Network Interfaces and User Credentials.

Finally, this document discusses UEFI Driver porting considerations and UEFI Driver optimization techniques for Intel IA-32-, Intel x64- and Intel® Itanium®-based platforms, as well as EFI Byte Code (EBC) platform types supported by the *UEFI Specification*.

The UEFI Driver Writers Guide uses the names defined by the EDK II open source project when referring to the various platform types.

- **IA32**—Intel IA-32 platforms
- **X64**—Intel® 64 platforms
- **IPF**—Intel® Itanium®-based platforms
- **EBC**—EFI Byte Code platforms

1.1.1 Assumptions

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

- Unified Extensible Firmware Interface Specification, Version 2.3.1.
- The EDK II is an open-source build environment project that is under constant development. EDK II not only provides the build environment, but also provides build tools and source code for firmware and drivers.

Note: The EDK II project on <http://www.tianocore.org> is under active development, often on a daily basis. Make sure to use a validated release of the UDK2010 for all UEFI Driver development.

- The UDK2010 Developer's Kit, referred to in this guide as the *UDK2010*, contains EDK II validated common-core sample code. The open-source UDK2010 is a stable build of the EDKII project and has been validated on a variety of Intel platforms, operating systems and application software. The open-source UDK2010 is available for download at www.tianocore.org
- The UDK2010 supports UEFI Driver development using the following operating system environments: Microsoft Windows™, UNIX and like systems and MAC OS X®. Refer to <http://www.tianocore.org> for a complete list of current development operating systems.
- The UDK2010 supports the development of UEFI Drivers using several families of compilers including those from Microsoft®, Intel and GCC. Refer to <http://www.tianocore.org> for a complete list of currently supported compilers.

1.2

Organization of this document

This document is not intended to be read front to back. Use it more as a cookbook for developing and implementing drivers. The following table describes the organization of this document.

Table 1—Organization of the *UEFI Driver Writer's Guide*

Chapter		Description
1	Introduction	Introduction and list of references related to UEFI Driver development.
2	Checklist	Checklist, or basic recipe, for UEFI Driver development.
3	Foundation	Foundation and terms related to UEFI Driver development.
4–17	Common Features	Recommendations for features common to most UEFI Driver types. Many of these features are optional and inclusion of them depends on the requirements for a specific UEFI Driver.
18–21	Industry Standard Busses	Recommendations for UEFI Drivers that manage controllers on Industry standard buses such as PCI, USB, SCSI and SATA.
22–27	Console and OS Boot Devices	Recommendations for UEFI Drivers that produce protocols that directly or indirectly provide services for a UEFI Boot Manager to initialize consoles and boot a UEFI conformant operating system from a boot device. This includes text consoles, serial ports, graphical consoles, mass storage devices, network devices and boot devices not defined by the <i>UEFI Specification</i> .
28–29	CPU Specific	Special considerations for IPF and EBC platforms.
30–32	Build/Release	Best practices for building, testing, debugging and distributing UEFI Drivers.
A	File Templates	Source file templates for UEFI Drivers, Protocols, GUIDs, and Library Classes
B	EDK II Drivers	Table of UEFI Driver features found in EDK II driver implementations.
C	Glossary	Glossary of terms used in this guide.

1.3 Related information

This chapter contains references to specifications, publications and tools referenced by other sections of this guide that may be useful in the development of UEFI Drivers. Find more information about UEFI tables, UEFI protocols, UEFI GUIDs, UEFI device types and UEFI status codes is in the *UEFI Specification*. This same information is also available from the Doxygen-generated help documents in the UDK2010 MdePkg. All source code examples in this guide follow the C coding style defined in the *EDK II C Coding Standard Specification*.

1.3.1 UEFI Specifications

Unified Extensible Firmware Interface, version 2.3.1, The UEFI Forum, 2010, <http://www.uefi.org>.

Find information about the differences between different versions of the *UEFI Specification* at <http://www.uefi.org>.

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

Microsoft Extensible Firmware Initiative FAT32 File System Specification, Version 1.03, Microsoft Corporation, December 6, 2000, <http://msdn.microsoft.com/en-us/windows/hardware/gg463080>

.

1.3.2 Industry Standard Bus Specifications

PCI Express Base Specification, Revision 2.1, PCI Special Interest Group, Hillsboro, OR, <http://www.pcisig.com/specifications>.

PCI Hot-Plug Specification, Revision 1.0, PCI Special Interest Group, Hillsboro, OR, <http://www.pcisig.com/specifications>.

PCI Local Bus Specification, Revision 3.0, PCI Special Interest Group, Hillsboro, OR, <http://www.pcisig.com/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).

Universal Serial Bus Revision 3.0 Specification bundle, USB Implementers Forum, Inc., 2011, <http://www.usb.org> (this bundle is referred to as USB Spec).

E-EDID EEPROM Specification, VESA, <http://www.vesa.org>

1.3.3 Other specifications

Advanced Configuration and Power Interface Specification, Revision 5.0, 2011,
<http://www.acpi.info>.

The Unicode Standard, Version 5.2, Unicode Consortium,
<http://www.unicode.org/versions/Unicode5.2.0>.

ISO 639-2:1998. Codes for the Representation of Names of Languages—Part2: Alpha-3 code, <http://www.iso.org>.

[RFC 4646] Tags for Identifying Languages, IETF, 2005,
<http://www.ietf.org/rfc/rfc4646.txt>.

Intel® 64 and IA-32 Architecture Software Developer’s Manual, Intel Corporation,
<http://www.intel.com/products/processor/manuals>.

Intel® Itanium® Architecture Software Developer’s Manual, vols. 1–4, Intel Corporation, <http://www.intel.com/design/itanium/manuals/iiasdmanual.htm>.

The current version of the manual includes Itanium® Processor Family System Abstraction Layer Specification.

A Formal Specification of Intel® Itanium® Processor Family Memory Ordering, Intel Corporation, <http://www.intel.com/design/itanium/downloads/251429.htm>.

Developer’s Interface Guide for Intel Itanium Architecture-based Servers (DIG64). 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>.

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

Harnessing the UEFI Shell: Moving the platform beyond DOS, Michael Rothman, Tim Lewis, Vincent Zimmer, and Robert Hale, ISBN 978-1-934053-14-0.

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

1.3.4 EDK II and UDK2010 Development Kit

UDK2010 Developer's Kit, <http://www.tianocore.org> (known hereafter as UDK2010).

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

EDK II User's Manual. <http://www.tianocore.org>

EDK II C Coding Standards Specification. <http://www.tianocore.org>

EDK II Build Specification. <http://www.tianocore.org>

EDK II Module Information File (INF) Specification. <http://www.tianocore.org>

EDK II Package Declaration File (DEC) Specification. <http://www.tianocore.org>

EDK II Platform Description File (DSC) Specification. <http://www.tianocore.org>

EDK II Flash Description File (FDF) Specification. <http://www.tianocore.org>

EDK II MdePkg Document. <http://www.tianocore.org>

Visual Forms Representation Programming Language document, Intel Corporation,
<http://www.tianocore.org>.

1.4 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 a 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 or to emphasize a critical term.

Italic In text, an ***Italic*** typeface can be used as emphasis to introduce a new term or to indicate the title of documentation, such as a user's manual or name of a specification.

Monospace Computer code, example code segments, pseudo code, 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).

2

UEFI Driver Implementation Checklist

The following is a checklist for implementing good, conformant, and efficient UEFI Drivers. References to sections of the guide that apply to each of the items in the checklist are provided so a UEFI Driver developer can easily determine the sections of the guide that apply to a specific UEFI Driver development task. The terminology used in this checklist is introduced in [Chapter 3](#).

When possible, copy an existing UEFI Driver with similar features and modify it to match the new UEFI Driver requirements. [Appendix B](#) contains a table of UEFI Drivers and features that each implements to help select an existing UEFI Driver.

Some UEFI drivers are ported from PC BIOS legacy option ROMs or EFI/UEFI Drivers based on previous releases of the *EFI/UEFI Specification*. While porting a driver from one environment to another is often done to save time and leverage resources, note that it requires careful attention to detail. Without a complete understanding of the target environment, the final driver can have remnants of the previous design that may degrade performance and functionality in the new environment.

1. Determine UEFI Driver Type

- o UEFI Driver Model ([Section 3.10](#) and [Chapter 6](#))
 - Must produce Driver Binding Protocol ([Chapter 9](#))
 - Device Driver ([Section 6.1](#), [Section 7.2](#), and [Chapter 9](#)) zw
 - Bus Driver ([Section 6.2](#), [Section 7.2](#), and [Chapter 9](#))
 - Hybrid Drive ([Section 6.3](#), [Section 7.2](#), and [Chapter 9](#))
 - Determine Optional UEFI Driver Model Features
 - Component Name 2 Protocol ([Section 7.1](#), [Section 7.2](#), [Chapter 11](#))
 - Component Name Protocol ([Section 7.1](#), [Section 7.2](#), [Chapter 11](#))
 - Driver Family Override Protocol ([Section 7.2.4](#) and [Chapter 15](#))
 - Driver Diagnostics 2 Protocol ([Section 7.1](#), [Section 7.2](#), [Chapter 13](#))
 - HII Packages ([Section 7.1](#), [Section 7.4](#), and [Chapter 12](#))
 - HII Config Access Protocol ([Section 7.1](#), [Section 7.5](#), and [Chapter 12](#))
 - Driver Configuration 2 Protocol ([Section 7.1](#) and [Chapter 12](#))
 - Driver Configuration Protocol ([Section 7.1](#) and [Chapter 12](#))
 - Driver Health Protocol ([Section 7.1](#), [Section 7.2.3](#), [Chapter 14](#))
 - Bus Specific Driver Override Protocol ([Chapter 17](#))

- Service Binding Protocol ([Chapter 10](#))
 - Service Driver ([Section 6.4](#) and [Section 7.9](#))
 - Root Bridge Driver ([Section 6.5](#) and [Section 7.10](#))
 - Initializing Driver ([Section 6.6](#) and [Section 7.8](#))
2. Determine Optional UEFI Driver Features
- Install an `Unload()` handler ([Section 7.6](#) and [Section 5.2.1.2](#))
 - HII Packages ([Section 7.1](#), [Section 7.4](#), and [Chapter 12](#))
 - HII Config Access Protocol ([Section 7.1](#), [Section 7.5](#), and [Chapter 12](#))
 - Driver Supported EFI Version Protocol ([Chapter 6](#), [Section 7.3](#), [Chapter 16](#))
 - Required for all plug in cards
 - Service Binding Protocol ([Chapter 10](#))
3. Identify the required UEFI supported CPU architectures
- IA32 ([Chapter 4](#))
 - X64 ([Chapter 4](#))
 - IPF ([Chapter 4](#) and [Chapter 28](#))
 - EFI Byte Code ([Chapter 4](#), [Section 4.4](#), [Section 18.6](#), and [Chapter 29](#))
4. Identify consumed I/O protocols
- PCI I/O Protocol to access a PCI Controller ([Chapter 18](#))
 - Always call `PciIo->Attributes()` ([Section 18.3.2](#))
 - Advertises dual address cycle capability
 - Save and enable attributes in `Start()`
 - Disable attributes in `Stop()`
 - DMA—Bus master write operations ([Section 18.5](#))
 - Must call `PciIo->Flush()`
 - DMA—Setting up with `PciIo->Map()` ([Section 18.5](#))
 - Do not use returned device address
 - Not all chipsets have 1:1 bus/system mappings
 - PCI Option ROM ([Section 18.7](#))
 - USB I/O Protocol to access a USB ([Chapter 19](#)) Device
 - SCSI I/O Protocol to access a SCSI Device ([Chapter 20](#))

- ATA Pass Thru Protocol to access a SATA Device ([Chapter 21](#))
5. Identify the boot related protocol(s) the UEFI Driver must produce
 - Keyboard ([Section 22.2](#))
 - Mouse
 - Tablet
 - Text Console ([Section 22.3](#))
 - Serial Port ([Section 22.4](#))
 - Graphics Console ([Chapter 23](#))
 - Mass Storage ([Chapter 24](#))
 - Network Controller ([Chapter 25](#))
 - Load File Protocol ([Chapter 27](#))
 - User Credential Provider ([Chapter 26](#))
 - USB Host Controller ([Section 19.1](#))
 - SCSI Host Controller ([Section 20.1](#))
 - ATA Host Controller ([Section 21.1](#) and [Section 20.1](#))
 6. Build UEFI Driver ([Chapter 30](#))
 7. Test and Debug UEFI Driver ([Chapter 31](#))
 - Use UEFI Shell to load and exercise functionality
 - Test all produced protocols
 - Test on multiple platforms
 - Pass UEFI SCT tests for the devices the UEFI Driver manages

2.1

Design and implementation of UEFI drivers

The following lists the basic steps a driver writer should follow when designing and implementing a UEFI driver. Note that this document assumes UEFI driver model drivers are being developed.

1. Determine the category of UEFI driver to be developed. The different categories are listed in [Table 2](#), below, and are described in more detail in [Chapter 6](#) of this document.

Note: *UEFI Drivers that follow the UEFI Driver Model are recommended because they enable faster platform boot times.*

2. Make sure the driver supports the unload service. This feature is strongly recommended for all drivers. [Section 7.6](#) describes the unload service.
3. Make sure the UEFI driver supports both the Component Name protocol and the Component Name2 protocol. It is strongly recommended that all drivers support both protocols.
4. Is the UEFI driver going to include configuration settings that the user can change? If so, the driver must support HII functionality. Note that the HII functionality replaces the Driver Configuration Protocol, which is now obsolete.

Table 2—Classes of UEFI drivers to develop

Class of Driver	See sections
Device driver	6.1
Bus driver that can produce one or all child handles	6.2.6
Bus driver that produces all child handles in the first call to <code>Start()</code>	6.2.7
Bus driver that produces at most one child handle in <code>Start()</code>	6.2.8
Bus driver that produces no child handles in <code>Start()</code>	6.2.9
Bus driver that produces child handles with multiple parent controllers	6.2.4
Hybrid driver that can produce one or all child handles	6.3 and 6.2.6
Hybrid driver that produces all child handles in the first call to <code>Start()</code>	6.3 and 6.2.7
Hybrid driver that produces at most one child handle in <code>Start()</code>	6.3 and 6.2.8
Hybrid driver that produces no child handles in <code>Start()</code>	6.3 and 6.2.9
Hybrid driver that produces child handles with multiple parent controllers	6.3 and 6.2.4
Service driver	6.4 and 7.9
Root bridge driver	6.5 and 7.10
Initializing driver	6.6 and 7.8

5. The UEFI driver must produce the Driver Diagnostics Protocols if the driver is going to support testing See [Chapter 13](#).
6. If the UEFI driver is a bus driver for a bus type that supports storage of UEFI drivers with the child devices, the Bus Specific Driver Override Protocol must be implemented by the bus driver. See [Chapter 17](#) of this guide.
7. A UEFI driver might not need to call an Exit Boot Service event. However, if the UEFI driver is going to require an Exit Boot Services event, then the driver must create an event of type Exit Boot Services. When the driver initializes, it creates the event, and when Exit Boot Services happens, the system calls the function that the driver produces. See [Chapter 7](#).
8. For runtime drivers, make sure the driver defines an event of type Set Virtual Address Map. This allows the driver to know where the memory map is located once the OS takes control. See [Chapter 7](#).

9. Identify the I/O-related protocols 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. For example, a console driver might have multiple binding protocols to allow for input from multiple devices. See [Chapter 9](#).
10. Identify all I/O-related protocols that the driver binding model must produce. Once the I/O-related protocols are known, make sure the driver creates a function with a single entry point for each protocol.
11. Implement the driver's entry point. See [Chapter 7](#).
12. Design the private context data structure. See [Chapter 8](#).
13. Implement all the services listed in the supported section of the Driver Binding Protocol. See [Chapter 9](#).

Table 3—Protocols produced by various devices

Device	Produces these I/O protocols
USB peripherals	USB I/O protocol
PCI adapter	PCI I/O protocol
Console devices	Simple input protocol Simple pointer protocol Graphics output protocol
Media devices	Block I/O protocol
SCSI, SCSI RAID, and Fiber Channel	Extended SCSI pass thru protocol Block I/O protocol
NIC (network interface controller)	The protocols produced by the NIC depends on the specific NIC Universal network driver interface (UNDI) protocol Network interface identifier protocol Managed network protocol (MNP) Simple network protocol (SNP)

Note: *The device path protocol is a data structure protocol, not a function call with a callable entry point. It is the UEFI driver's job to append the path of the devices it is controlling to the data structure. In other words, as part of producing the I/O protocol for each device, the driver builds the device path for that device.*

2.2

How to implement features in EDK II

The first column of the table below describes functions a typical driver performs. Column 2 briefly describes how each function is implemented in UEFI and references the chapter in this guide that specifically addresses each issue. This list of driver operations is not exhaustive.

Table 4—Mapping operations to UEFI drivers

Operation	Recommended UEFI method
Find devices that the driver supports while the driver is running	<p>Do not try to search the handle database specifically. Instead, allow the supported section of the driver binding protocol to do this operation.</p> <p>The supported section checks to see if the driver supports the device for the specified controller handle. The supported section uses the controller handle along with a partial device path, to check to see if the specific device is supported, and returns <i>supported</i>, <i>already started</i>, or <i>not-supported</i> for each device.</p>
Search devices that the driver supports	<p>Use shell applications, such as the <code>dh</code> (dump handle database) command or the drivers shell command.</p> <p>The <code>dh</code> command returns a list of all devices on the system. The drivers command returns a list of all drivers on the system. With the list of drivers, the <code>dh -d</code> command can be used to list the handles which that driver supports.</p>
Perform DMA	<p>Use the DMA-related services from the PCI I/O Protocol. See the PCI driver section (Chapter 18) of this guide.</p>
Access PCI configuration header	<p>Always use PCI I/O Protocol services to access the PCI configuration header. Never directly access I/O ports 0xCF8 or 0xCFC.</p> <p>See the PCI driver section (Chapter 18) of this guide.</p>
Access PCI I/O ports	<p>Always use PCI I/O Protocol services to access PCI I/O ports. Never use IN or OUT instructions.</p> <p>See the PCI driver section (Chapter 18) of this guide.</p>
Access PCI memory	<p>Use PCI I/O Protocol services to access PCI memory. Never use pointers to directly access memory-mapped I/O resources on a bus.</p> <p>See the PCI driver section (Chapter 18) of this guide.</p>
Hardware interrupts	<p>EDK II does not support legacy INT type hooking interrupts. Instead, UEFI drivers are expected to either perform block I/O, by which they must complete their I/O operation and poll their devices as required to complete it, or they can create a periodic timer event to get control and check the status of the devices under management.</p> <p>See the services section (Chapter 5) and the general driver guidelines section (Chapter 4) of this guide for more detail.</p>
Calibrated stalls	<p>Do not use hardware devices to perform calibrated stalls. Instead, use the <code>Stall()</code> service for short delays that are typically less than 10 ms. Use one-shot timer events for long delays that are typically greater than 10 ms. Use <code>SetTimer()</code> in conjunction with <code>CreateEvent()</code>, or <code>CreateEventEx()</code>, for longer delays. Do not use the <code>GetTime()</code> service for delays in UEFI drivers. Use it only to retrieve information. See the services section in this guide: Services that UEFI drivers commonly use.</p>

Operation	Recommended UEFI method
Get keyboard input from user	<p>Use the HII interface to accept keyboard input from the user. The HII engine displays forms to the user in which the user can answer questions or provide input. The forms themselves are defined in the VFR standard.</p> <p>Note that console-related services, such as Simple Text Input Protocol and Simple Text Output Protocol can be replaced with or supplemented by HII functionality and forms.</p> <p>Note that the Driver Configuration Protocol service is obsolete and has been replaced with HII functionality.</p>
Display text	<p>Use the HII interface to display text to the user. The HII engine displays forms to the user and allows querying of the user. The forms themselves are defined by the VFR programming language and IFR specification.</p> <p>Note that console-related services, such as Simple Text Input Protocol and Simple Text Output Protocol, can be replaced with or supplemented by HII functionality and forms.</p> <p>Also, note that the Driver Configuration Protocol service is obsolete and has been replaced with HII functionality.</p>
Diagnostics	Implement both the Driver Diagnostics Protocol and the Driver Diagnostics2 Protocol. See Chapter 13 of this guide.
Flash utility	UEFI drivers should not try to reprogram a flash device. Typically, a flash device is reprogrammed by a standalone application, such as a UEFI utility.
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 a UEFI driver touched the controller 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 Chapter 7 .

3

Foundation

UEFI employs several key concepts as cornerstones of understanding for UEFI Drivers. These concepts are defined in the *UEFI Specification*. Programmers new to UEFI should find the following introduction to a few of UEFI's key concepts helpful as they study the *UEFI Specification*.

The basic concepts covered in the following sections include:

- Basic programming model
- Objects managed by UEFI-conformant firmware
- UEFI system table
- Handle database
- Protocols
- UEFI images
- Events
- Task priority levels
- Device paths
- UEFI driver binding model
- Platform initialization
- Boot manager and console management
- EDK II libraries

As each concept is discussed, the related application programming interfaces (APIs) are identified along with references to the related sections in the *UEFI Specification*.

One of the components available from the EDK II open source project and distributed with the UDK2010 releases is the UEFI Shell; a command line interface with useful commands for development and testing of UEFI drivers and UEFI applications. The UEFI Shell also provides commands to help illustrate many of the basic concepts described in the sections that follow. These useful UEFI Shell commands are identified as each concept is introduced. The UEFI Shell is an open source project at <http://www.tianocore.org> where documents providing details on all of the available commands can be found.

3.1 Basic programming model

Common questions about UEFI include:

- How are programs in UEFI implemented?
- What makes UEFI programming different from an operating system?

- What makes UEFI different from other firmware environments?
- In particular, what is the programming model for a UEFI Driver?

Key points about writing UEFI-conformant drivers are that:

- UEFI Drivers are relocatable PE/COFF images whose format is defined by the Microsoft Portable Executable and Common Object File Format Specification.
- UEFI Drivers may be compiled for any of the CPU architectures supported by the *UEFI Specification*.
- UEFI Drivers run on a single CPU thread.
- The driver support infrastructure does not extend beyond the boot processor.
- Drivers sit above some interfaces (for example, bus abstractions) and below other interfaces: They are both consumers and producers. The *UEFI Specification* defines the interfaces and they are extensible.
- Each driver is expected to cooperate with other drivers, other modules and the underlying core services.
- The communicating modules bind together to create stacks of cooperating drivers to accomplish tasks.
- Inter-module communication is enabled via interfaces known as protocols and via events.
- Tables provided at invocation provide access to core services.
- The operating environment is non-preemptive and polled. There are no tasks per se. Instead, modules execute sequentially.
- There is only one interrupt: the timer. This means that data structures accessed by both in-line code and timer-driven code must take care to synchronize access to critical paths. This is accomplished via privilege levels.

3.2 Objects managed by UEFI-based firmware

Objects of several differing types are managed through the services provided by UEFI. [Figure 1](#) shows the various object types. The most important objects for UEFI drivers are the following:

- UEFI 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 UEFI system table provides access to all services provided by UEFI. The system table also provides 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, are introduced in the following sections.

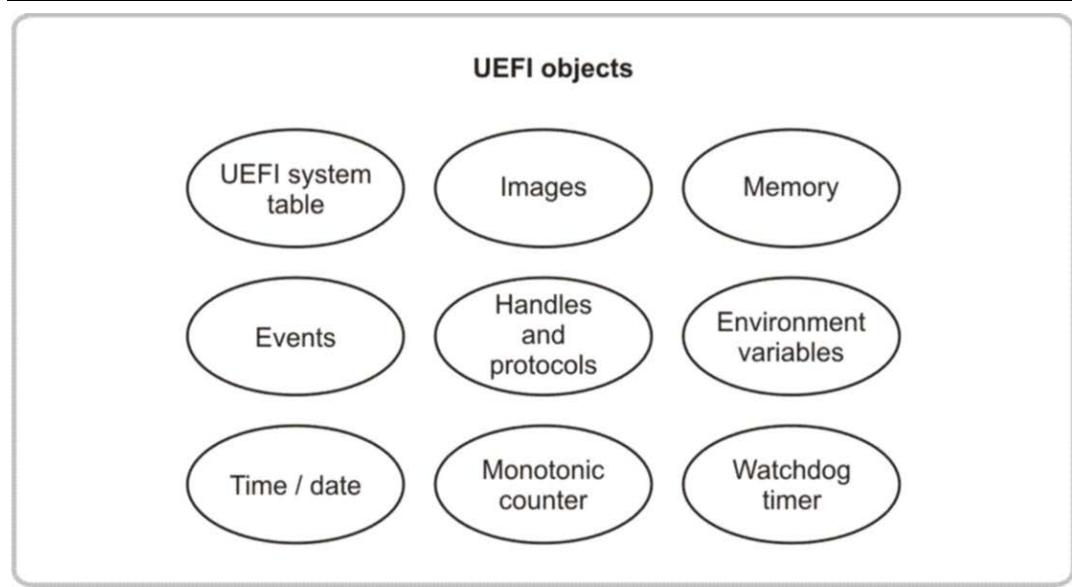


Figure 1—Object managed by UEFI-based firmware

3.3 UEFI system table

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

- UEFI boot services
- UEFI runtime services
- Services provided by protocols

Two of the data fields in the UEFI system table, UEFI boot services and UEFI runtime services, are accessed through the UEFI boot services table and the UEFI runtime services table, respectively. The number and type of services that are available from these two tables are fixed for each revision of the *UEFI Specification*. The UEFI boot services and UEFI runtime services are defined in the *UEFI Specification*. The specification also describes the common uses of these services by UEFI drivers.

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 Specification*). Protocol services are typically used to provide software abstractions for devices such as consoles, mass storage devices 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 Specification* defines over 30 different protocols, and various implementations of UEFI firmware. UEFI drivers may produce additional protocols to extend the functionality of a platform.

3.4 Handle database

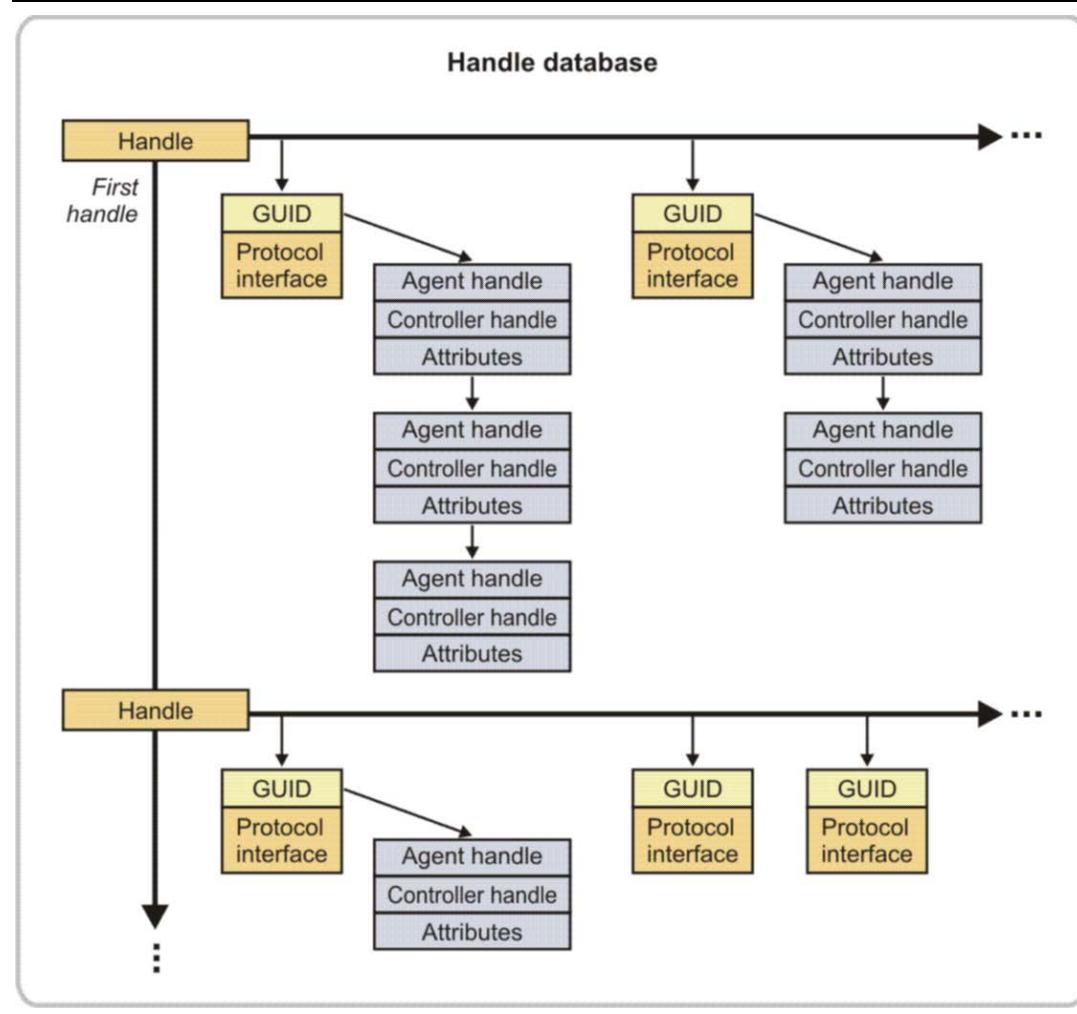
The handle database is composed of objects called handles and protocols. *Handles* are a collection of one or more protocols and *protocols* are data structures named by a GUID. The data structure for a protocol may contain data fields, services, both or none at all.

At reset, the Handle Database is empty. During platform initialization, the system firmware, UEFI conformant drivers and UEFI applications create handles and attach one or more protocols to the handles. Information in the handle database is “global” and accessible by any executable UEFI image.

The handle database is a list of UEFI handles and is the central repository for the objects maintained by UEFI-based firmware. Each UEFI handle identified by a unique handle number 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 a GUID attach to a UEFI handle and determine the handle type. A UEFI handle may represent components like:

- Executable images such as UEFI drivers and UEFI applications
- Devices such as network controllers and hard drive partitions
- UEFI services which are accessed as drivers such as EFI Decompress and the EBC Interpreter

The [following figure](#) shows a portion of the handle database. In addition to the handles and protocols, a list of objects is associated with each protocol. The handle database uses this list to track which agents are consuming which protocols. This information is critical to the operation of UEFI drivers. It is what allows UEFI drivers to be safely loaded, started, stopped and unloaded without resource conflicts.

**Figure 2—Handle database**

[Figure 3—Handle types](#), 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 appear throughout the document.

There is only one handle database and all handles reside in it. Services that manage the Handle database do not distinguish handle types. Handles are differentiated by the types of protocols associated with each handle.

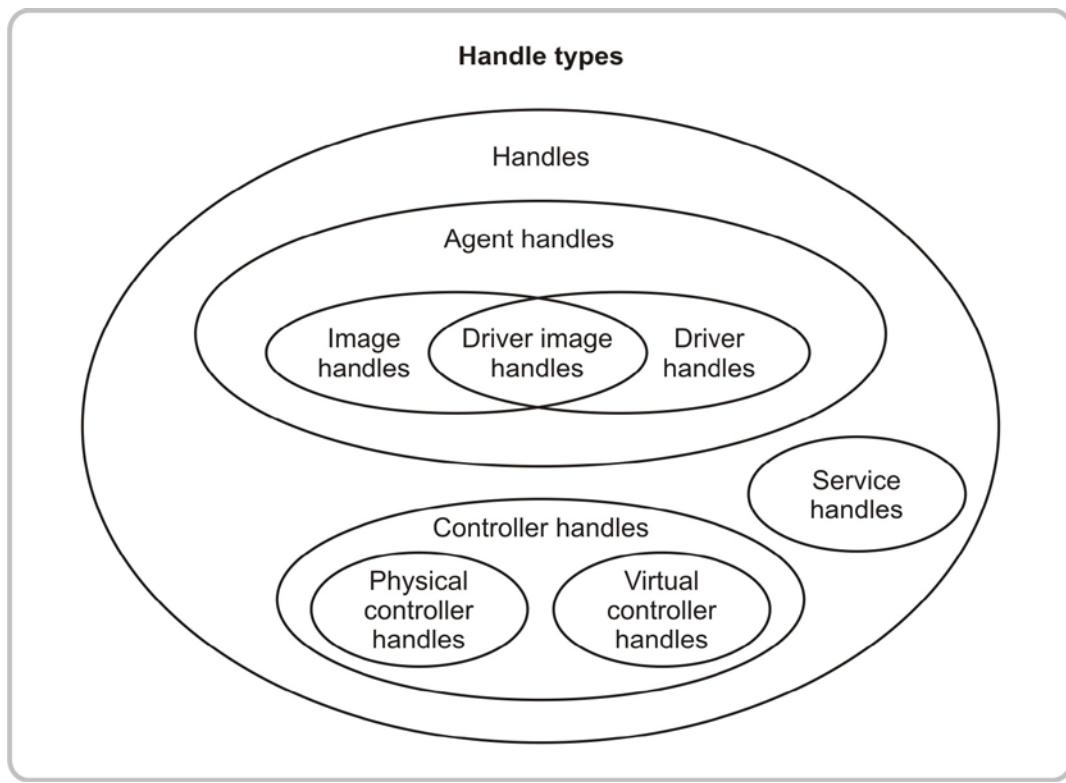


Figure 3—Handle types

The following table describes the types of handles shown above. The *UEFI Specification* provides detailed information on these types of handles, the protocols they support, and the different driver types. Note that HII handles are considered service handles.

Table 5—Description of handle types

Type of handle	Description
Image handle	This is the handle for the UEFI Driver image loaded into memory. It supports the Loaded Image Protocol.
Driver handle	Supports all UEFI protocols. The most common protocols are the Driver Binding Protocol, the two Component Name Protocols and the two Driver Diagnostics Protocols.
Driver image handle	This is a handle that has the attributes of both an Image Handle and a Driver Handle. It is the handle for a UEFI Driver image loaded into memory. It supports the Loaded Image Protocol, and it supports the UEFI Driver related protocols.
Agent handle	Some of the UEFI driver model-related services in the <i>UEFI Specification</i> use this term. An <i>agent</i> is a UEFI component that can consume a protocol in the handle database. An <i>agent handle</i> is a general term that can represent an image handle, a driver handle or a driver image handle.

Type of handle	Description
Controller handle	<p>A controller handle represents a console or boot device that is present in the platform. 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 <i>not</i> 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 <i>UEFI Specification</i> include the following:</p> <ul style="list-style-type: none"> • Console Services: These have been replaced or supplemented by HII functionality. These protocols include the 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: Extended SCSI Pass Thru Protocol and SCSI I/O Protocol. • Graphics Services: Graphics Output Protocol.
Device handle	Used interchangeably with <i>controller handle</i> .
Bus controller handle	A Controller Handle managed by a bus driver or a hybrid driver-producing 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—a child handle—is the only distinction between a controller handle and a bus controller handle.
Child handle	This is a Controller Handle created by a bus driver or a hybrid driver. 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 representing a physical device that must support the Device Path Protocol. See the <i>UEFI Specification</i> .
Virtual controller handle	A controller handle representing a virtual device and not supporting the Device Path Protocol.

Type of handle	Description
Service handle	<p>A handle referencing certain types of tasks such as decompression or HII forms display. It can interface with other drivers, but does not relate to hardware or file management. This type of handle is not used for the Loaded Image Protocol, the Driver Binding Protocol or the Device Path Protocol. Instead, this type of handle 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 <i>UEFI Specification</i> include:</p> <ul style="list-style-type: none"> • HII functionality • Platform Driver Override Protocol • Unicode Collation Protocol • Boot Integrity Services Protocol. • Debug Support Protocols. • Decompress Protocol (optional). To give developers more flexibility, the EDK II open source project provides several decompression algorithms. • EFI Byte Code (EBC) Protocol

3.5 GUIDs

A UEFI programming environment provides software services through the UEFI Boot Services Table, the UEFI Runtime Services Table, and Protocols installed into the handle database. Protocols are the primary extension mechanism provided by the *UEFI Specification*. Protocols are named using a GUID.

A GUID is a unique 128-bit number that is a globally unique identifier (a universally unique identifier, or UUID). Each time an image, protocol, device, or other item is defined in UEFI, a GUID must be generated for that item. The example below shows the structure definition for an `EFI_GUID` in the EDK II along with the definition of the GUID value for the EFI Driver Binding Protocol from the *UEFI Specification*.

```

///+
/// 128 bit buffer containing a unique identifier value.
/// Unless otherwise specified, aligned on a 64 bit boundary.
///
typedef struct {
    UINT32 Data1;
    UINT16 Data2;
    UINT16 Data3;
    UINT8 Data4[8];
} GUID;

///
/// 128-bit buffer containing a unique identifier value.
///
typedef GUID           EFI_GUID;

///
/// The global ID for the Driver Binding Protocol.
///

```

```
#define EFI_DRIVER_BINDING_PROTOCOL_GUID \
{ \
    0x18a031ab, 0xb443, 0x4d1a, {0xa5, 0xc0, 0xc, 0x9, 0x26, 0x1e, 0x9f, 0x71} \
}
```

Example 1—EFI_GUID data structure in EDK II

TIP: New GUID values can be generated using the **GUIDGEN** utility shipped with Microsoft™ compilers, or the **uuidgen** command under Linux. Other GUID generation utilities may be found using internet searches.

Protocol services are registered in the handle database using the GUID name of the Protocol and Protocol services are discovered by looking up Protocols in the handle database using the GUID name associated with the Protocol to perform the lookup operation.

UEFI fundamentally assumes that a specific GUID exposes a specific protocol interface (or other item). Because a protocol is “named” by a GUID (a unique identifier), there should be no other protocols with that same GUID. Be careful when creating protocols to define a new, unique GUID for a new protocol.

Put another way, the GUID forms a contract: If the UEFI Driver finds a protocol with a particular GUID, it may assume that the contents of the protocol are as specified for that protocol. If the contents of the protocol are different, the driver that published the protocol is assumed to be in error.

In some ways, GUIDs are can be viewed as contracts. If a UEFI Driver looks up a protocol with a certain GUID, the structure under the GUID is well defined. If the GUID is duplicated, this 1:1 mapping breaks. If a GUID is copied and applied to a new protocol, the users of the old protocol call the new protocol expecting the old interfaces or vice versa. Either way, the results are never good.

Caution: *There are improper practices to create new GUID values. For example, cutting and pasting an existing GUID, hand-modifying an existing GUID, or incrementing/decrementing fields in a GUID creates the opportunity to introduce a duplicate GUID. These practices can cause **catastrophic failures**. Typically, a system containing a duplicate GUID may inadvertently find the new protocol and think that it is another protocol, which mostly likely crashes the system. Another possible failure is a data-loss failure caused when a duplicated GUID is a data-handling GUID (such as a disk I/O, file-system or NVRAM-handling GUID). Always use a GUID generator utility to create new GUIDs.*

TIP: Bugs caused by duplicate GUIDs are typically very difficult to root cause and many developers do not check the GUID when debugging. If the root cause for a hang has not been found in a reasonable amount of time, check to make sure the GUID for each relevant protocol is unique.

3.6 Protocols and handles

The extensible nature of UEFI is built, to a large degree, around protocols. Protocols serve to enable communication between separately built modules, including drivers.

Drivers create protocols consisting of two parts. The body of a protocol is a C-style data structure known as a protocol interface structure, or just “interface”. The interface typically contains an associated set of function pointers and data structures.

Every protocol has a GUID associated with it. The GUID serves as the name for the protocol. The GUID also indicates the organization of the data structure associated with the GUID. Note that the GUID is not part of the data structure itself.

The example below shows a portion of the Component Named 2 Protocol definition from the UEFI Driver Model chapter of the *UEFI Specification*. Notice that the protocol data structure contains two functions and one data field.

```
/// Global ID for the Component Name Protocol
///
#define EFI_COMPONENT_NAME2_PROTOCOL_GUID \
{0x6a7a5cff, 0xe8d9, 0x4f70, { 0xba, 0xda, 0x75, 0xab, 0x30, 0x25, 0xce, 0x14 } }

typedef struct _EFI_COMPONENT_NAME2_PROTOCOL  EFI_COMPONENT_NAME2_PROTOCOL;

///
/// This protocol is used to retrieve user readable names of drivers
/// and controllers managed by UEFI Drivers.
///
struct _EFI_COMPONENT_NAME2_PROTOCOL {
    EFI_COMPONENT_NAME2_GET_DRIVER_NAME      GetDriverName;
    EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME  GetControllerName;

    ///
    /// A Null-terminated ASCII string array that contains one or more
    /// supported language codes. This is the list of language codes that
    /// this protocol supports. The number of languages supported by a
    /// driver is up to the driver writer. SupportedLanguages is
    /// specified in RFC 4646 format.
    ///
    CHAR8                                     *SupportedLanguages;
};
```

Example 2—Protocol structure in EDK II

Protocols are gathered into a single database. The database is not “flat.” Instead, it allows protocols to be grouped together. Each group is known as a handle, and the handle is also the data type that refers to the group. The database is thus known as the handle database. Handles are allocated dynamically. Protocols are not required to be unique in the system, but they must be unique on a handle. In other words, a handle may not be associated with two protocols that have the same GUID.

3.6.1 Protocols are produced and consumed

Protocols enable inter-module communication in UEFI. To enable this communication, one of the modules must create or “produce” the protocol. Other modules (including drivers) may then use or “consume” the protocol.

Drivers are both consumers and producers of protocols. For example, a UEFI Driver for a SCSI Host Controller on a PCI bus consumes the PCI I/O Protocol and produces the SCSI Host Controller Protocols.

The initial producer of the protocol must "create" the protocol. The protocol structure must be allocated from memory (allocated either statically in the program or via a memory allocation operation). The protocol must then be initialized by filling in its contents. This almost always involves filling in the function pointers declared in the protocol structure. In other words, to produce a protocol is to declare its functionality and publish that functionality to the handle database (so other drivers can find and use that declaration).

Although it is legal to store data in a protocol, this is **strongly discouraged** for data items that may change over time. It is not a safe way to store dynamic data. Instead, functions that provide get/set operations (as in object-oriented programming) are safer and more extensible. The producer then uses `InstallMultipleProtocolInterfaces()` (as defined in the Boot Service chapter of the *UEFI Specification*) or similar to install the protocol into the handle database and make the protocol available to others.

The consumer has a somewhat simpler task. The consumer looks up the protocol in the handle database by GUID. With service protocols, for which there is only one instance in the entire handle database, the consumer can use the `LocateProtocol()` service. For protocols that may be present on multiple handles in the handle database, the `LocateHandleBuffer()` service can be used to locate the set of handles that support a specified protocol. The consumer can then use the `OpenProtocol()` service to lookup a protocol on a specific handle.

It is possible that the consumer is invoked before the producer. In this case, the consumer can request it be notified when new instances of the protocol are created. This is accomplished using the `RegisterProtocolNotify()` service.

Any UEFI image can use protocols during boot time. However, after `ExitBootServices()` is called, the handle database is no longer available to the image.

A complete description of all the services used to manage the handle database and produce and consume protocols appears in [Chapter 5](#).

3.6.2 Protocol interface structure

The [following figure](#) shows a single handle and protocol from the handle database produced by a UEFI driver. The protocol is composed of a GUID and a protocol interface structure.

Many times, the UEFI driver that produces a protocol interface maintains 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. A UEFI driver may produce one protocol or many protocols depending on the driver's complexity.

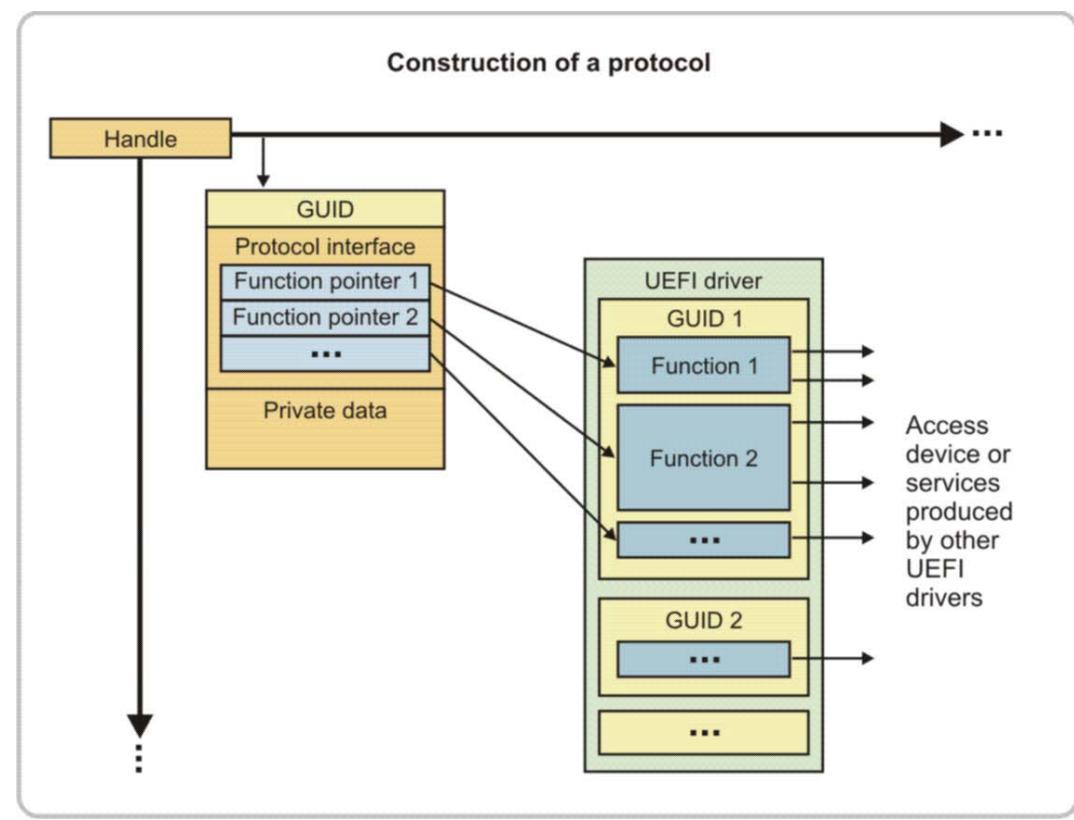


Figure 4—Construction of a protocol

3.6.3 Protocols provided in addition to the *UEFI Specification*

Not all protocols are defined in the *UEFI Specification*. For example, the EDK II, like other developer's kits, includes additional protocols that are not part of the *UEFI Specification*. These additional protocols are necessary to provide all of the functionality in a particular implementation but they are not defined in the current *UEFI Specification* because they do not present an external interface—a requirement to support booting of an OS or writing of a 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 EDK II that are not part of the *UEFI Specification*:

- Print 2 Protocol
- [MdeModulePkg/Include/Protocol/Print2.h](#)
- Deferred Procedure Call Protocol
- [MdeModulePkg/Include/Protocol/Dpc.h](#)
- VGA Mini Port Protocol
- [IntelFrameworkModulePkg/Include/Protocol/VgaMiniPort.h](#)

UEFI Drivers and UEFI OS Loaders should not depend on these types of protocols because they are not guaranteed to be present in every UEFI-conformant firmware implementation. UEFI Drivers and UEFI OS Loaders should depend only on protocols defined in the current *UEFI Specification* and protocols required by platform design guides (i.e. *DIG64*). The extensible nature of UEFI allows each platform to design and add its own special protocols. Use these protocols to expand the capabilities of UEFI and provide access to proprietary devices and interfaces congruent with the rest of the UEFI architecture.

3.6.4 Multiple protocol instances

Multiple protocols are installed on the same handle if the protocols provide services related to that one handle. There are several handle types. The most common are image handles and device handles. For example, if there are multiple I/O services for a single device that are abstracted through multiple protocols, then multiple protocols must be installed onto the handle for that device.

A handle may have many protocols attached to it. However, it may have only one protocol of each GUID name. 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 unique to that PCI device, including the location and size of the UEFI-conformant 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 Protocols on its driver image handle, yet when the Component Name Protocols' `GetDriverName()` function is called, each handle returns the unique name of the driver that owns that image handle. The `GetDriverName()` function on the USB bus driver handle returns “USB bus driver” for the English language, but the `GetDriverName()` function on the PXE driver handle returns “PXE base code driver.”

3.6.5 Tag GUID

A protocol may be nothing more than a GUID with no associated data structure. This GUID is also known as a *tag GUID*. Such a protocol can be useful, for example, to mark a device handle as special in some way or allow other UEFI images to find the device handle easily by querying the system for the device handles with that protocol GUID attached.

3.7 UEFI images

There are different types of UEFI images, but all UEFI images contain a PE/COFF header that defines the format of the executable code. The PE/COFF image header follows the format defined by the *Microsoft Portable Executable and Common Object File Format Specification*. The code can be for IA32, X64, IPF, or EBC. The header defines the processor type and the image type. Refer to the UEFI Image section of the

Overview chapter in the *UEFI Specification* for definitions of the processor types and the following three image types:

- UEFI applications
- UEFI boot services drivers
- UEFI runtime drivers

UEFI images are loaded and relocated into memory with the boot service `LoadImage()`. There are several supported storage locations for UEFI images, including:

- Expansion ROMs on a PCI card
- System ROM or system flash
- A media device such as a hard disk, floppy, CD-ROM, DVD, FLASH drive
- 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. This allows the UEFI image to be placed anywhere in system memory. The Boot Service `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 a UEFI image is loaded with `LoadImage()`, the image can be started with a call to `StartImage()`. The header for a UEFI image contains the address of the entry point called by `StartImage()`. The entry point always receives the following two parameters:

- The image handle of the UEFI image being started
- A pointer to the UEFI system table

The image handle and pointer allow the UEFI image to:

- 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 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.

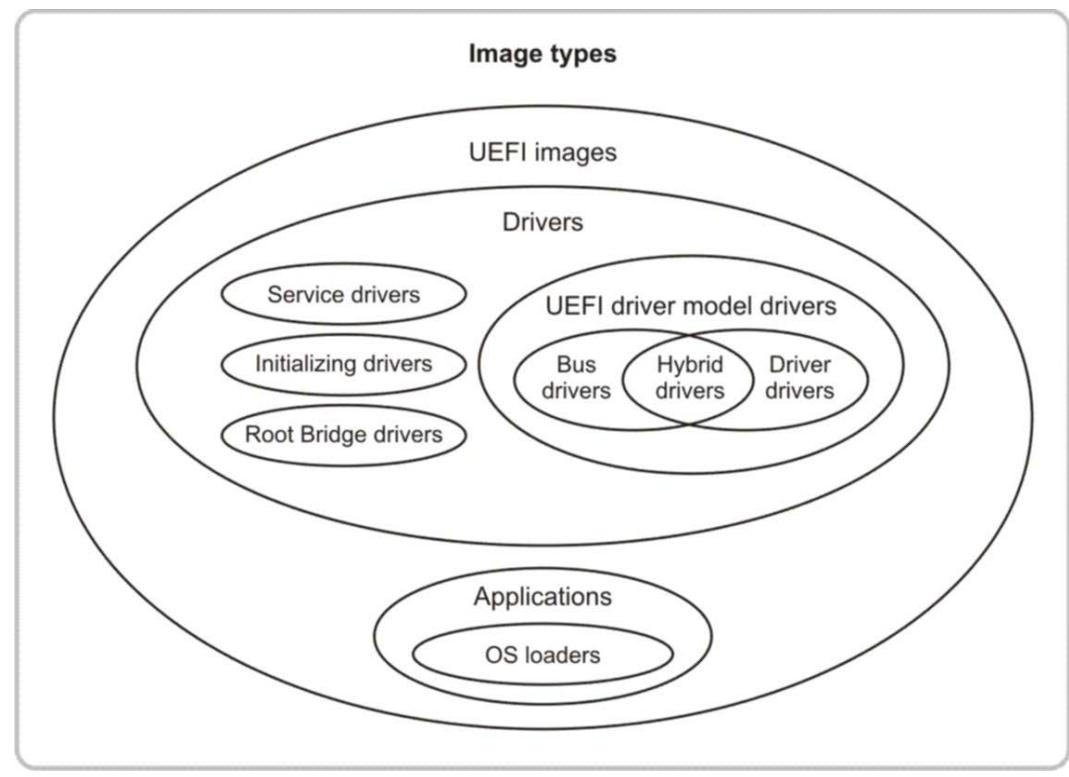


Figure 5—Image types

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

Table 6—Description of image types

Type of image	Description
Application	A 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 ExitBootServices() to transfer control of the platform from the firmware to an operating system.

Driver	A UEFI image of type <code>EFI_IMAGE_SUBSYSTEM_BOOT_SERVICE_DRIVER</code> or <code>EFI_IMAGE_SUBSYSTEM_RUNTIME_DRIVER</code> . If this image returns <code>EFI_SUCCESS</code> , then the image is not unloaded. If the image returns an error code other than <code>EFI_SUCCESS</code> , 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 <code>ExitBootServices()</code> .
Service driver	A driver that produces one or more protocols on one or more new service handles and returns <code>EFI_SUCCESS</code> 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 described in the UEFI Driver Model chapter of the <i>UEFI Specification</i> . 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 following 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 <code>Start()</code> service of the Driver Binding Protocol is called. Instead, it only adds additional I/O protocols to existing controller handles.
Bus driver	A driver following 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 <code>Start()</code> 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 <code>Start()</code> service of the Driver Binding Protocol adds I/O protocols to existing handles and creates child handles.

3.7.1 Applications

A UEFI application starts execution at its entry point and then executes until it returns from its entry point or it calls the `Exit()` boot service function. When done, the image is unloaded from memory. It does not stay resident. Some examples of common UEFI applications include the following:

- UEFI Shell
- UEFI Shell Applications
- Flash utilities
- Diagnostic utilities

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

3.7.1.1 OS loader

The *UEFI Specification* details a special type of UEFI application called an *OS boot loader*. It is a 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 platform firmware frees all of its boot time services and boot time drivers, leaving only the runtime services and runtime drivers.

3.7.2 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 platform firmware, the boot manager, and UEFI applications may load drivers.

3.7.2.1 Boot service drivers

Boot 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 `ExitBootServices()` is called.

3.7.2.2 Runtime drivers

Runtime drivers are loaded in memory marked as `EfiRuntimeServicesCode`. They allocate their data structures from memory marked as `EfiRuntimeServicesData`. These types of memory are preserved after `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 alternative calling mechanism is application-specific.

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. The EDK II does include an UNDI driver. UEFI Drivers for Network Interface

Controllers (NICs) are discussed in detail in [Chapter 25](#). Other runtime drivers are not common and are not discussed in this guide.

3.7.2.2.1

Be rigorous when implementing runtime drivers

Implementing and validating runtime drivers is much more difficult than implementing and validating boot service drivers. The difficulties occur because UEFI supports the translation of runtime services and runtime drivers from a physical addressing mode to a virtual addressing mode. For example, a pointer might not have the same value in the physical address space as it might in the virtual address space. Getting that translation, or mapping, correct is very difficult because if even a single pointer translation is missed, the OS may crash or hang if the runtime driver is called and a code path that accesses that pointer is used. Debugging runtime services provides by UEFI Drivers at OS runtime is more difficult than debugging UEFI Drivers in the pre-boot environment. Since some code paths are executed infrequently, careful code review and extensive validation of runtime drivers is strongly recommended. Also, there are no utilities to perform such translations automatically. Each piece of data and memory allocation must be inspected manually to determine if it needs to be adjusted. That in itself can be an error-prone process. Additionally, if another driver writer tries to adjust the code, that writer might not be aware of each piece of data or memory allocation that must be adjusted.

There are best practices to help perform these translations. However, great care must be taken to follow the recommended practices and UEFI requirements rigorously. Many of the requirements for runtime drivers are listed in the *UEFI Specification*. Make sure they are well understood. Of particular importance are the sections on runtime services, and specifically, virtual memory.

3.8

Events and task priority levels

Events are another type of object that is managed through UEFI services. They provide synchronous or asynchronous call back upon a particular occurrence. They can be created and destroyed and are either in the waiting state or the signaled state. A 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.

UEFI supports polled drivers, not interrupts. Because UEFI does not support interrupts, it can present a challenge to driver writers who are used to an interrupt-driven driver model.

The most common use of events by a UEFI driver is the use of timer events that allow drivers to poll a device periodically. The figure below shows the different types of events supported in UEFI, as well as the relationships between those events.

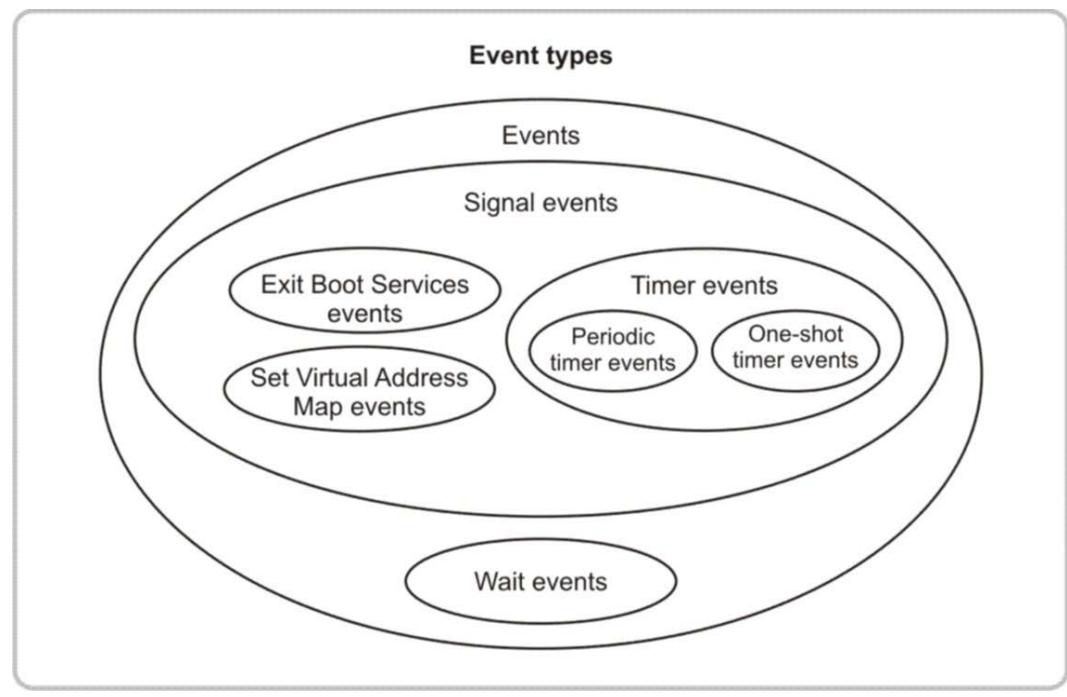


Figure 6—Event types

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

Table 7—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 <code>ExitBootServices()</code> 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 <code>ExitBootServices()</code> 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 UEFI runtime service <code>SetVirtualAddressMap()</code> 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 <code>SetVirtualAddressMap()</code> is called.

Type of events	Description
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 time 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 UEFI are listed in the table below.

Table 8—Task priority levels defined in UEFI

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 UEFI. (Not usable by drivers)

TPLs serve two purposes:

- Define the priority in which notification functions are executed
- Create locks

3.8.1 Defining priority

Notification functions at higher priorities can interrupt the execution of notification functions executing at a lower priority.

The mechanism for defining the priority (in which notification functions are executed), 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 is executed first.

3.8.2 Creating locks

It is possible for the code in normal context and the code in interrupt context (i.e. notification functions) to access the same data structure. This is because UEFI *does* support a single timer interrupt. This access can cause issues if the updates to a shared data structure are not atomic. A 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 blocks even the one timer interrupt. However, care must be taken to minimize the amount of time that the system executes at [`TPL_HIGH_LEVEL`](#). All timer-based events are blocked during this time and any driver requiring periodic access to a device is prevented from accessing its device. See the Boot Services chapter of the *UEFI Specification* for more information on Task Priority Levels and [Section 5.1.4](#) of this guide for examples on how Task Priority Levels can be used to create and manage locks.

3.8.3 Using callbacks

The calls to create an event take two important parameters: the callback and the parameter pointer.

The callback function is invoked when the event occurs. Using callbacks appropriately is not difficult—as long as the following rules are followed:

- The parameter pointer can point to any static (not on the stack) structure. The parameter pointer is used to provide state information for the event invocation. The parameter pointer is particularly useful if multiple events must be handled by the same callback.
- The callback function, when invoked, may only assume its priority level, its parameter pointer, and that it has a stack. It must derive all context from the parameter pointer and the static data left in its module. This makes writing callbacks somewhat more challenging than normal driver code.

3.8.3.1 Debugging callbacks

Debugging callbacks is a little like debugging interrupt handlers in that one is not always sure when a callback is invoked. Most normal debugging facilities function as expected in callbacks.

There can be a temptation to write one's driver as a series of callbacks. This is not recommended since normal code is easier to debug, and managing a large number of the context structures addressed by parameter pointers becomes difficult to maintain.

TIP: Minimize the use of callbacks. Only use a callback when an operation cannot be implemented as part of UEFI Driver initialization or through a protocol services provided by the UEFI Driver.

3.9 UEFI device paths

UEFI defines a Device Path Protocol that is attached to device handles in the handle database. The Device Path Protocol helps 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 UEFI-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. It also 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.

The following two examples show the declaration of the PCI device path node which combined the generic UEFI Device Path Header with the PCI-device-specific fields *Function* and *Device*.

```
/**  
 * This protocol can be used on any device handle to obtain generic path/location  
 * information concerning the physical device or logical device. If the handle does  
 * not logically map to a physical device, the handle may not necessarily support  
 * the device path protocol. The device path describes the location of the device  
 * the handle is for. The size of the Device Path can be determined from the  
 * structures  
 * that make up the Device Path.  
 */  
typedef struct {  
    UINT8 Type;           // /< 0x01 Hardware Device Path.  
    // /< 0x02 ACPI Device Path.  
    // /< 0x03 Messaging Device Path.  
    // /< 0x04 Media Device Path.  
    // /< 0x05 BIOS Boot Specification Device Path.  
    // /< 0x7F End of Hardware Device Path.  
  
    UINT8 SubType;        // /< Varies by Type  
    // /< 0xFF End Entire Device Path, or  
    // /< 0x01 End This Instance of a Device Path and start a new  
} DevicePathHeader;
```

```

///< Device Path.

UINT8 Length[2];    ///< Specific Device Path data. Type and Sub-Type define
                     ///< type of data. Size of data is included in Length.

} EFI_DEVICE_PATH_PROTOCOL;

```

Example 3—Device Path Header

```

///
/// PCI Device Path.
///
typedef struct {
    EFI_DEVICE_PATH_PROTOCOL           Header;
    ///
    /// PCI Function Number.
    ///
    UINT8                            Function;
    ///
    /// PCI Device Number.
    ///
    UINT8                            Device;
} PCI_DEVICE_PATH;

```

Example 4—PCI Device Path

Device paths are designed to be position-independent by not using pointer values for any field. 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. See [Example 2](#) in this section.

The following table lists the types of device path nodes that are defined in the Device Path Protocol chapter of the *UEFI Specification*.

Table 9—Types of device path nodes defined in *UEFI 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 supports multiple directory levels.
BIOS Boot Specification (BBS) device path node	Used to describe a device that has a type that follows the BIOS Boot Specification, such as floppy drives, 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 types may have to be created. The vendor-defined nodes use a GUID to distinguish new device path nodes.

Careful design is required when choosing 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 if there are system configuration changes in other parts of the platform. 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 defined in the EFI Device Path Display Format Overview section of the *UEFI Specification*. This optional feature allows developers to view device paths in a readable form using the UEFI shell. The UEFI Shell also provides a method to perform a hex dump of a device path.

The example below shows some example device paths. 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 5—Device Path Examples

3.9.1

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 data structures. In general, UEFI Drivers are not required to parse or understand the beginning of the device path. They usually only need to understand the device path node associated with the specific controller the UEFI Driver is managing and, potentially, the device path node associated with child controllers the UEFI Driver may generate by appending a new device path node to the end of the device path from the parent controller.

- 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.
- For a child device, bus drivers usually just append a single device path node to that 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, consider a SCSI Bus Driver that produces child handles for the mass storage devices on a SCSI channel. This UEFI Driver builds a device path for each mass storage device. The device path is constructed by appending a SCSI device path node to the device path of the SCSI channel. The SCSI device path node simply contains the Physical Unit Number (PUN) and Logical Unit Number (LUN) of the SCSI mass storage device.

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 Protocol defined by the *UEFI Specification* to help manage device paths. The **MdePkg** in the EDK II also provides a Device Path Library with many useful functions and macros to manage device paths.

3.9.2 IPF Considerations for device path data structures

Individual device paths nodes may be any length, and each device path node in a complete device path starts immediately after the previous device path node. This means that device path nodes inside of a full device path may not start on a naturally aligned boundary. This can cause problems for CPU architectures that do not support unaligned memory accesses such as IPF. A device path node that is not a multiple of 8 bytes in length may cause a device path node that follows to be unaligned. Implementing source code that manages device paths requires some special techniques to guarantee that the source code is portable to all the CPU architectures supported by the *UEFI Specification*.

TIP: Be careful when using device paths. Make sure an alignment fault is not generated.

See [Chapter 4](#) in this guide for more information about architecture-specific considerations. Refer to [Chapter 28](#) for IPF platform porting considerations.

3.9.3 Environment variables

Device paths are also used when certain environment variables are built and stored in non-volatile storage. There are a number of environment variables defined in the Boot Manager chapter of the *UEFI 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-conformant operating systems manage these environment variables as operating systems are installed and removed from a platform.

3.10 UEFI driver model

The Overview and UEFI Driver Model chapters of the *UEFI Specification* define 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 their loading into memory from their starting and stopping. The table below lists the series of UEFI driver model-related protocols that are used to accomplish this separation.

Table 10—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 binding model requires this protocol.
Service Binding Protocols	Provides a mechanism that allows protocols to support more than one consumer. UEFI Drivers that are required to produce protocols that need to be available to more than one consumer produce both the Driver Binding Protocol and a Service Binding Protocol.
Driver Supported EFI Version Protocol	Provides information on the version of the <i>UEFI Specification</i> to which the UEFI Driver conforms. The version information follows the same format as the version field in the EFI System Table.
Driver Family Override Protocol	Provides a mechanism for a UEFI Driver to express UEFI Driver specific version information among a family of UEFI Drivers that are used by ConnectController() to select the best driver to manage a specific controller.
Driver Health Protocol	Provides services that allow a UEFI Driver to express messages associated with the health status of a controller, suggest repair operations, and request configuration changes required to place the controller in a usable state.
HII Config Access Protocol	Provides services to retrieve and save configuration data for a controller managed by a UEFI Driver. Also provides a service that allows a setup browser to inform a UEFI Driver when specific setup browser actions are performed.
HII Packages	Allows a UEFI Driver to register strings, fonts, images, keyboard mappings, and setup forms related to the configuration operations required for UEFI Driver managed controllers.
Component Name 2 Protocol	Provides functions for retrieving a human-readable name of a driver and the controllers that a driver is managing using language codes defined by RFC 4646.
Driver Diagnostics 2 Protocols	Provides functions for executing diagnostic functions on driver managed devices using RFC 4646 defined language codes.
Component Name Protocol	Provides functions for retrieving a human-readable name of a driver and the controllers that a driver is managing using language codes defined by ISO 639-2. This protocol is only required by a UEFI Driver that must be compatible with platforms that support only UEFI 2.0 or EFI 1.10. This protocol has been replaced by the Component Name 2 Protocol.

Driver Diagnostics Protocols	Provides functions for executing diagnostic functions on driver managed devices using language codes defined by <i>ISO 639-2</i> . This protocol is only required by a UEFI Driver specifically compatible with platforms supporting only UEFI 2.0 or EFI 1.10. This protocol has been replaced by the Driver Diagnostics 2 Protocol.
Driver Configuration Protocol	Provides functions that allow users to configure devices a driver is managing using language codes defined by <i>ISO 639-2</i> . It also provides services to place a device into a default configuration. This protocol is only required by a UEFI Driver specifically compatible with platforms supporting only UEFI 2.0 or EFI 1.10. This protocol has been replaced with HII functionality.

The new protocols are registered on the driver's image handle. HII packages are registered in the HII database. In the UEFI driver model, the main goal of the driver's entry point is to install these protocols, register HII packages, 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, additional instances of the Driver Binding Protocol are installed on new handles. These new handles may also optionally support the additional protocols listed in [Table 10](#) above.

The UEFI driver model follows the organization of physical/electrical architecture by defining three basic types of UEFI boot time drivers:

- Device drivers
- Bus drivers
- Hybrid drivers, which have characteristics of both a device driver and a bus driver

Device drivers and bus drivers are distinguished by the operations they perform in the **Start()** and **Stop()** services of the 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 become evident; the following sections discuss these two driver types.

3.10.1 Device driver

The **start()** service of a device driver installs protocol(s) directly onto the controller handle that was passed into the **start()** service. The protocol(s) installed by the device driver use 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 USB device uses the service of the USB I/O Protocol, and a device driver for a PCI controller uses the services of the PCI I/O Protocol. In other words, the PCI I/O Protocol is consumed by a driver for a PCI option ROM card. 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 directly or indirectly by UEFI-conformant system firmware to boot an operating system.

It does not make sense to write device drivers for devices that cannot be used to boot a platform. The following table provides the list of standard I/O protocols that the *UEFI Specification* defines for different classes of devices. If multiple protocols are listed, that does not necessarily mean that all the protocols must be produced. Please see later sections of the guide and the *UEFI Specification* for details on which protocols are required and which are optional.

Table 11—I/O protocols produced in the Start() function for different device classes

Class of device	Protocol(s) created in the Start section of the driver
Block Oriented Device	<code>EFI_BLOCK_IO2_PROTOCOL</code> <code>EFI_BLOCK_IO_PROTOCOL</code> <code>EFI_STORAGE_SECURITY_COMMAND_PROTOCOL</code>
File System	<code>EFI_SIMPLE_FILE_SYSTEM_PROTOCOL</code>
Non block oriented or file system based boot device	<code>EFI_LOAD_FILE_PROTOCOL</code>
LAN	Universal Network Driver Interface (UNDI) <code>EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL</code> <code>EFI_SIMPLE_NETWORK_PROTOCOL</code> <code>EFI_MANAGED_NETWORK_PROTOCOL</code> <code>EFI_VLAN_CONFIG_PROTOCOL</code> <code>EFI_BIS_PROTOCOL</code>
Graphics Display	<code>EFI_GRAPHICS_OUTPUT_PROTOCOL</code> <code>EFI_EDID_DISCOVERED_PROTOCOL</code> <code>EFI_EDID_ACTIVE_PROTOCOL</code>
Text Console	<code>EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL</code>
Character based I/O device	<code>EFI_SERIAL_IO_PROTOCOL</code>
Keyboard	<code>EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL</code> <code>EFI_SIMPLE_TEXT_INPUT_PROTOCOL</code>
Mouse	<code>EFI_SIMPLE_POINTER_PROTOCOL</code>
Tablet	<code>EFI_ABSOLUTE_POINTER_PROTOCOL</code>
USB Host Controller	<code>EFI_USB2_HC_PROTOCOL</code> <code>EFI_USB_HC_PROTOCOL</code>
SCSI Host Controller	<code>EFI_EXT_SCSI_PASS_THRU_PROTOCOL</code> <code>EFI_SCSI_PASS_THRU_PROTOCOL</code>

SATA Controller	<code>EFI_ATA_PASS_THRU_PROTOCOL</code>
Credential Provider for User Authentication	<code>EFI_USER_CREDENTIAL2_PROTOCOL</code>

The fundamental definition of a UEFI device driver is that it does not create any child handles. This difference distinguishes a device driver from a bus driver.

The definition of a device driver can be confusing because it is often necessary to write a driver that creates child handles. This necessity makes the driver a bus driver by definition, even though the driver may not be managing a hardware bus in the classical sense (such as a PCI, SCSI, USB, or Fibre Channel bus).

Even though a device driver does not create child handles, the device managed by the device driver could still become 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*. The protocol is consumed by the USB bus driver. The USB bus driver 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.

3.10.2 Bus driver

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

Just as with a device driver, the `start()` function of a bus driver consumes the parent bus I/O abstraction(s) and produces 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 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().
- A bus driver that creates child handles across multiple Start() calls is very useful because it may reduce the platform boot time. It allows a few child handles, or even a single child handle, to be created across multiple calls to Start(). 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 the *UEFI Specification*).
- Load drivers from option ROMs, if present. The PCI bus driver is currently the only bus driver that loads from option ROMs.

Some common examples of UEFI bus drivers include:

- **PCI Bus Driver:** Creates a child handle for PCI controllers, either directly attached to a PCI Root Bridge, or attached to a PCI Root Bridge through one or more PCI to PCI Bridges. The Device Path Protocol includes `PCI()` device path nodes.
- **USB Bus Driver:** Creates a child handle for USB devices, either directly attached to a USB Root Port, or attached to a USB Root Port through one or more USB Hubs. The Device Path Protocol includes `USB()` device path nodes.
- **SCSI Bus Driver:** Creates a child handle for SCSI devices attached to a SCSI channel. The Device Path Protocol includes `SCSI()` device path nodes.
- **SATA Bus Driver:** Creates a child handle for SATA devices attached to a SATA ports. The Device Path Protocol includes `SATA()` device path nodes.

Because bus drivers are defined as drivers that produce child handles, there are some other drivers 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 two or more physical video outputs. Graphics drivers do not extend the Device Path Protocol.

3.10.3 Hybrid driver

A hybrid driver manages and enumerates a bus controller. Its `Start()` function creates one or more child handles and installs protocols into the child handles. Its `Start()` function also installs protocols onto the handle for the bus controller itself.

3.11 Service Drivers

A service driver does not manage any devices and does not produce any instances of the `EFI_DRIVER_BINDING_PROTOCOL`. It is a simply a driver that produces one or more protocols on one or more new service handles in the handle database. 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 installed, leaving the driver resident in system memory. Some example service drivers in the `MdeModulePkg` in the EDK II include:

- `MdeModulePkg/Universal/Acpi/AcpiTableDxe`
- `MdeModulePkg/Universal/DebugSupportDxe`
- `MdeModulePkg/Universal/DevicePathDxe`
- `MdeModulePkg/Universal/EbcDxe`
- `MdeModulePkg/Universal/HiiDatabaseDxe`
- `MdeModulePkg/Universal/PrintDxe`
- `MdeModulePkg/Universal/SetupBrowserDxe`
- `MdeModulePkg/Universal/SmbiosDxe`

3.12 Root Bridge Driver

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 or root devices in a platform. The driver must install the Device Path Protocol onto a physical controller handle because the root bridge controllers or root devices represent physical devices. An example root bridge driver, `PcAtChipsetPkg/PciHostBridgeDxe`, is shown in the EDK II. This driver also installs the PCI Root Bridge I/O Protocol—the protocol abstraction for a PCI Bus. This protocol is used by a bus driver for the PCI Bus to enumerate the PCI controllers attached to the PCI root bridge.

A driver for a root device may produce a protocol that is more directly usable as a console or boot device. For example, a Serial I/O Protocol for a serial device that is not attached to an industry standard bus type supported by the *UEFI Specification*, or a Block I/O Protocol for a block-oriented media device that is not attached to an industry standard bus type supported by the *UEFI Specification*.

3.13 Initializing Driver

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. The EDK II does not currently include examples of UEFI initializing drivers.

3.14 UEFI Driver Model Connection Process

All UEFI Drivers that adhere to the UEFI Driver Model follow the same basic procedure. When the driver is loaded, it installs a Driver Binding Protocol on the image handle from which it was loaded. It may also update a pointer to the `Unload()` service of the Loaded Image Protocol and install the Component Name 2 Protocol and the Component Name Protocol, if needed, so its name is visible to any operator. The UEFI Driver then exits from the entry point with a return status of `EFI_SUCCESS`, leaving the UEFI Driver resident in system memory.

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

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

The Driver Binding Protocol is available on the driver's image handle after the entry point is exited. Later on when the system is "connecting" drivers to devices, the driver's Driver Binding Protocol `supported()` service is called.

The `Supported()` service is passed a controller handle. The `Supported()` function quickly examines the controller handle to see if it represents a device that the driver knows how to manage. If so, it returns `EFI_SUCCESS`. The system then starts 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.

A platform connects the devices in a platform with the drivers available in the platform. This connection process appears complex at first, but as the process continues, it becomes evident that the same basic procedure is used over and over again 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 understood. This knowledge is fundamental to designing new UEFI Drivers.

The UEFI boot service `ConnectController()` demonstrates the flexibility 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()`.

3.14.1 ConnectController()

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

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

```
/**  
 * Connects one or more drivers to a controller.  
  
 * @param ControllerHandle      The handle of the controller to which driver(s) are  
 *                               to be connected.  
 * @param DriverImageHandle     A pointer to an ordered list handles that support  
 *                               The EFI_DRIVER_BINDING_PROTOCOL.  
 * @param RemainingDevicePath   A pointer to the device path that specifies a child  
 *                               of the controller specified by ControllerHandle.  
 *                               If TRUE, then ConnectController() is called  
 *                               recursively until the entire tree of controllers  
 *                               below the controller specified by ControllerHandle  
 *                               have been created. If FALSE, then the tree of  
 *                               controllers is only expanded one level.  
  
 * @retval EFI_SUCCESS          1) One or more drivers were connected to  
 *                               ControllerHandle.  
 *                               2) No drivers were connected to ControllerHandle,  
 *                               But RemainingDevicePath is not NULL, and it is  
 *                               an End Device Path Node.  
 * @retval EFI_INVALID_PARAMETER ControllerHandle is NULL.  
 * @retval EFI_NOT_FOUND        1) There are no EFI_DRIVER_BINDING_PROTOCOL  
 *                               Instances present in the system.  
 *                               2) No drivers were connected to ControllerHandle.  
  
 */  
typedef  
EFI_STATUS  
(EFIAPI *EFI_CONNECT_CONTROLLER)(  
    IN  EFI_HANDLE           ControllerHandle,  
    IN  EFI_HANDLE           *DriverImageHandle,    OPTIONAL  
    IN  EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath, OPTIONAL  
    IN  BOOLEAN              Recursive  
);
```

Example 6—`ConnectController()` UEFI Boot Service

The connection is a two-phase process:

1. Construct an ordered list of driver handles from highest to lowest priority.
2. Attempt to connect the drivers to a controller in priority order from highest to lowest.

The following table lists the steps for phase one; *driver connection precedence rules*. Much of this information is in the *UEFI Specification* where the UEFI boot service `ConnectController()` is discussed.

Table 12—Connecting controllers: Driver connection precedence rules

Step	Type of override	Description

1	Context override	<p>The parameter <i>DriverImageHandle</i> is an ordered list of handles that support the EFI_DRIVER_BINDING_PROTOCOL. The highest priority image handle is the first element of the list, and the lowest priority image handle is the last element of the list. The list is terminated with a NULL image handle.</p> <p>This parameter is usually NULL and is typically used only to debug new drivers from the UEFI Shell. These drivers are placed at the top of the ordered list of driver handles.</p>
2	Platform driver override	<p>If an EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL instance is present in the system, the GetDriver() service of this protocol is used to retrieve an ordered list of image handles for <i>ControllerHandle</i>. From this list, the image handles found in rule (1) above are removed. The first image handle returned from GetDriver() has the highest precedence, and the last image handle returned from GetDriver() has the lowest. The ordered list is terminated when GetDriver() returns EFI_NOT_FOUND. It is legal for no image handles to be returned by GetDriver(). There can be, at most, a single instance in the system of the EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL. If there is more than one, then the system behavior is not deterministic.</p> <p>The EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL is optional and, if present, is provided with the platform firmware. This protocol is typically provided when a platform needs to guarantee that a specific UEFI Driver be used to manage a specific controller, which is typically only required for controllers that are integrated into the platform.</p>
3	Driver family override	<p>The list of available driver image handles can be found by using the boot service LocateHandle() with a <i>SearchType</i> of ByProtocol for the GUID of the EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL. From this list, the image handles found in rules (1), and (2) above are removed. The remaining image handles are sorted from highest to lowest based on the value returned from the GetVersion() function of the EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL associated with each image handle.</p> <p>The EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL is optional and is typically produced by UEFI Drivers associated with a family of controllers. When multiple versions of a UEFI Driver for a family of controllers are present in a platform, the UEFI Driver needs to determine which version of the UEFI Driver is best suited to manage a specific controller in the family of controllers.</p>

4	Bus specific driver override	<p>If there is an instance of the <i>EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL</i> attached to <i>ControllerHandle</i>, then the <i>GetDriver()</i> service of this protocol is used to retrieve an ordered list of image handles for <i>ControllerHandle</i>. From this list, the image handles found in rules (1), (2), and (3) above are removed. The first image handle returned from <i>GetDriver()</i> has the highest precedence, and the last image handle returned from <i>GetDriver()</i> has the lowest precedence. The ordered list is terminated when <i>GetDriver()</i> returns <i>EFI_NOT_FOUND</i>. It is legal for no image handles to be returned by <i>GetDriver()</i>.</p> <p>In practice, this precedent option allows the UEFI drivers that are stored in a PCI Option ROM of a PCI adapter to manage that specific PCI adapter, even if drivers with higher versions are available from PCI Option ROMs on other PCI adapters. This rule exists to make sure that if a particular UEFI Driver on a PCI adapter only works with the hardware on that specific PCI adapter, then a UEFI Driver from a different PCI adapter is not to be used to manage it. If an IHV does not like this precedence rule, the Driver Family Override Protocol can be implemented to override this behavior.</p>
5	Driver binding search	<p>The list of available driver image handles can be found by using the boot service <i>LocateHandle()</i> with a <i>SearchType</i> of <i>ByProtocol</i> for the GUID of the <i>EFI_DRIVER_BINDING_PROTOCOL</i>. From this list, the image handles found in rules (1), (2), (3), and (4) above are removed. The remaining image handles are sorted from highest to lowest based on the <i>Version</i> field of the <i>EFI_DRIVER_BINDING_PROTOCOL</i> instance associated with each image handle.</p> <p>In practice, this sorting means that a PCI adapter, for example, that does not have a UEFI driver in its PCI Option ROM is managed by the driver with the highest <i>Version</i> number.</p>

Phase two of the connection process checks each driver in the ordered list to see if it supports the controller. This check calls the ***Supported()*** service of the driver's Driver Binding Protocol and passes in the ***ControllerHandle*** and the ***RemainingDevicePath***. If successful, the ***Start()*** service calls the Driver Binding Protocol 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 if they also require exclusive access to the parent I/O protocol.

Use this type of connection process 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 UEFI initialization. Use it to connect consoles, devices required to load drivers from the driver list, and to connect devices required for the boot options to be processed by the boot manager.

3.14.2 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 drivers connect, 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 UEFI driver version 0x12 is soft loaded through the UEFI Shell, nothing changes 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.

The drivers now 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 IHV can override this logic by implementing the Driver Family Override Protocol.

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

3.14.3 DisconnectController()

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

3.15 Platform initialization

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

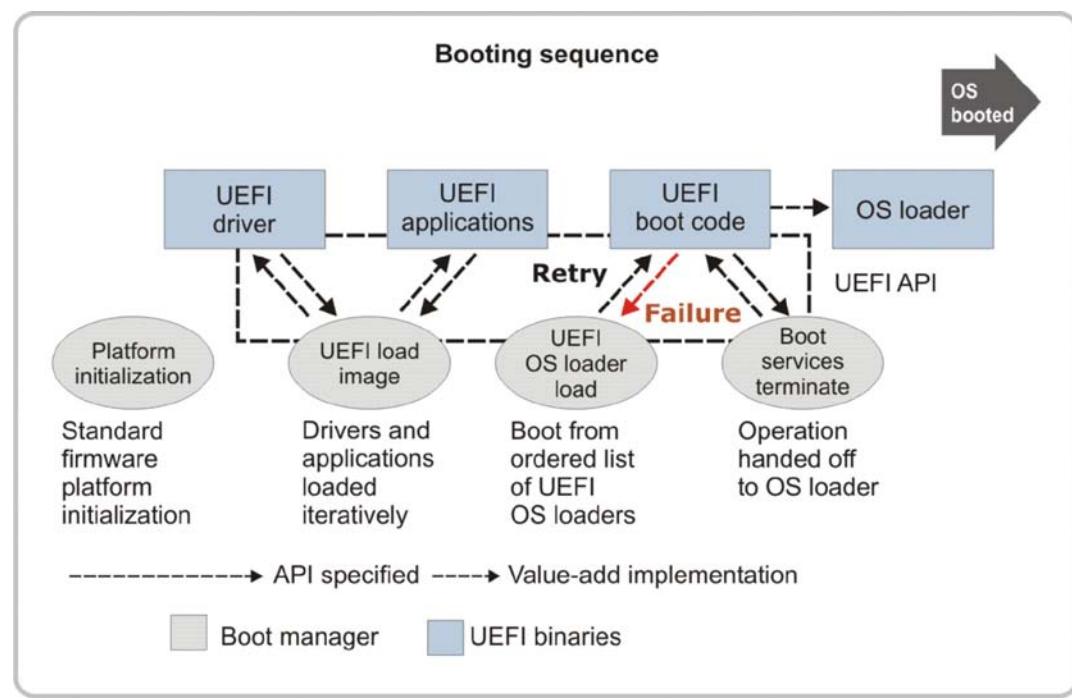


Figure 7—Booting sequence for UEFI operational model

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

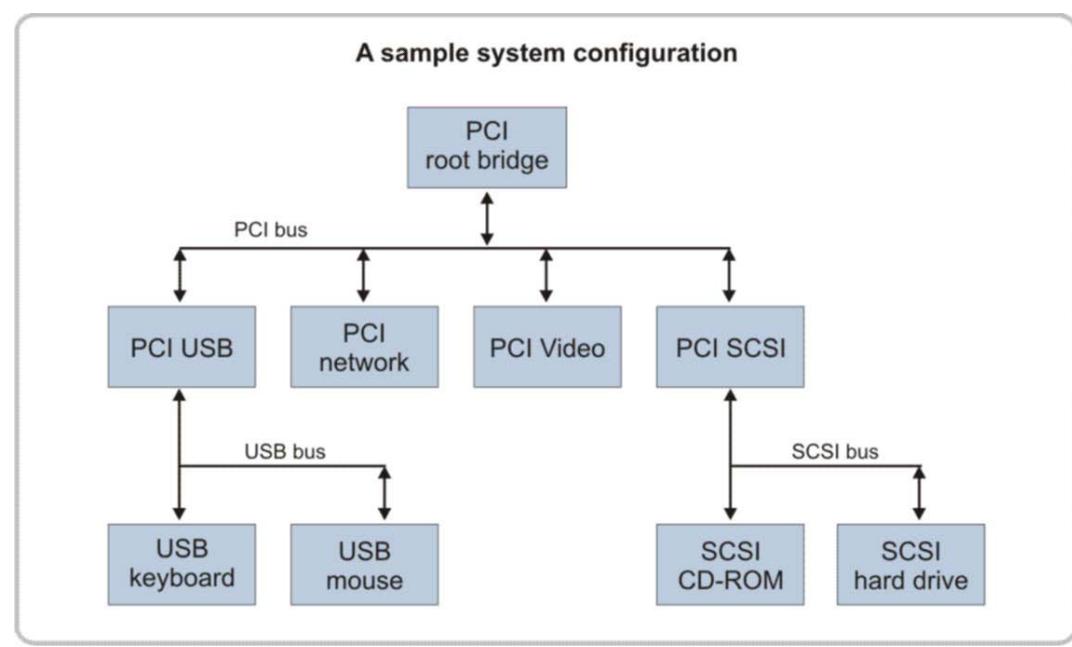


Figure 8—Sample system configuration

During platform initialization, early in the boot process, the platform creates handles and install the EBC Protocol and the Decompression Protocol(s) in the handle database. These service protocols are needed to run UEFI drivers that may be compressed or compiled using an EBC compiler. The Compression Algorithm Specification chapter of the *UEFI Specification* defines the `EFI_DECOMPRESS_PROTOCOL`, which defines the standard compression algorithm for use with UEFI Drivers stored in PCI Option ROMs.

For example, a portion of the handle database as viewed with the `dh` UEFI Shell command might look like the example below. Handle `6` supports the EBC Protocol. Handle `9` is an image handle for a UEFI Service Driver. That UEFI Device Driver installed the `EFI_DECOMPRESS_PROTOCOL` onto a new handle. The handle created is handle `A`.

```

...
6: Ebc
...
9: Image(Decompress)
A: Decompress

```

3.15.1 Connecting PCI Root Bridges

During UEFI-conformant firmware initialization by the platform, the system typically uses the service `LoadImage()` to load a root bridge driver for the root device. One common example is a PCI root bridge driver.

Like all drivers, as it loads, UEFI firmware creates a handle in the handle database and attaches 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. Instead, it immediately uses its knowledge about the platform architecture to create handles for each PCI root bridge

As viewed using the `dh` UEFI Shell command below, a portion of the handle database shows a single PCI root bridge. Some platforms, such as data center servers, will have more than one PCI root bridge.

A PCI root bridge driver installs the `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` and an `EFI_DEVICE_PATH_PROTOCOL` onto a new 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.

```
...
B: Image(PcatPciRootBridge)
C: 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 usually require access to the boot devices to complete an OS boot operation. Boot devices must have a Device Path Protocol that represents the unique name of the boot device. The Device Path Protocol for a boot device attached to a PCI Bus would start with a single ACPI node `Acpi(HID, UID)` or `Acpi(HID, UID, CID)`. This node also points the OS to the place in the ACPI name space where the ACPI description of the PCI root bridge is stored. The `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` provides PCI functions that are used by the PCI bus driver that is described in next section.

3.15.2 Connecting the PCI bus

Platform initialization continues by loading the PCI bus driver. As the driver's entry point is executed, the PCI bus driver installs the Driver Binding Protocol and potentially the Component Name Protocols.

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 started. It contains one new driver image handle with the Loaded Image Protocol, Driver Binding Protocol, and Component Name2 Protocol. Because this driver does follow the UEFI Driver Model, no new controller handles are produced when the driver is loaded and started. They are not produced until the driver is connected.

```
...
14: Image(PciBus) Driver Binding ComponentName
...
```

Later in the platform initialization process, UEFI-conformant firmware uses `ConnectController()` to attempt to connect the PCI root bridge controller(s) (handle #14 hex, as shown in the example above). The system has several priority rules for determining what driver to try first, but in this case it searches the handle database for

driver handles (handles with the Driver Binding Protocol). The search finds handle #14 and call the Driver Binding Protocol `Supported()` service, passing in *controller handle* #14. The PCI bus driver requires the Device Path Protocol and PCI Root Bridge I/O Protocol to be started, so the `Supported()` service returns `EFI_SUCCESS` when those two protocols are found on handle #14. After receiving `EFI_SUCCESS` from the `Supported()` service, `ConnectController()` then calls the Driver Binding Protocol `start()` service with the same controller handle #14.

Due to the PCI Bus Driver, the `Start()` service uses the PCI Root Bridge I/O Protocol functions to enumerate the PCI bus and discover all PCI devices. For each PCI device/function that the PCI bus driver discovers, it creates a child handle and installs 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 bus driver also copies the device path from the parent PCI root bridge device handle and appends a new PCI device path node `Pci(Dev|Func)`. In cases where the PCI bus driver discovers a PCI-to-PCI bridge, the devices below the bridge are added as children to the bridge. In these cases, extra PCI device path nodes are added for each PCI-to-PCI bridge 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 that:

- Nine PCI devices were discovered.
- The PCI device on handle #1B has an option ROM with a 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.

```
.
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)) Driver Binding
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))
.
.
```

3.15.3 Connecting consoles

At this point during the platform initialization, the firmware has not initialized or configured a “console” device that allows user input. This absence is often because a PCI device, waits for the PCI bus driver to provide device handles for the console(s).

Most UEFI conformant platforms follow a console connection strategy to connect the consoles in a manner consistent with that of the platform. This ensures that the platform is able to display messages to all of the selected consoles through the standard UEFI mechanisms. Initially, this includes platform initialization and

informational screens, and later (during setup), HII functionality and forms. Prior to this point, platform messages, if any, are conveyed through platform-specific methods.

Note: During initialization, the platform needs to connect console devices to the driver. HII functionality is about displaying configurable information to the user, which happens after consoles are initialized and after an HII compatible setup engine is invoked. UEFI Drivers should never directly access console devices except for the few UEFI driver related services that explicitly allow user interaction. In most cases, UEFI drivers use HII infrastructure to present information to users.

3.15.4 Console drivers

UEFI consoles drivers may include one or more of the following:

- Text console devices
- Graphical console devices
- Keyboards
- Mice
- Serial ports

Some systems may provide custom console devices. The following table shows examples of console related UEFI Drivers from the EDK II. These UEFI Drivers may be carried by the platform firmware or in standard containers for UEFI Drivers such as PCI Option ROMs.

Table 13—UEFI console drivers

Class of driver	Type of driver	Driver name	Description and example
USB Console	USB host controller driver	UhciDxe EhciDxe XhciDxe	Consumes the PCI I/O Protocol and produces the USB 2 Host Controller Protocol. 25: Image(EhciDxe) DriverBinding ComponentName2 ComponentName
	USB bus driver	UsbBusDxe	Consumes the USB Host Controller 2 Protocol and produces the USB I/O Protocol. 26: Image(UsbBusDxe) DriverBinding ComponentName2 ComponentName
	USB keyboard driver	UsbKbDxe	Consumes the USB I/O Protocol and produces the Simple Input Ex Protocol and Simple Input Protocol. 27: Image(UsbKbDxe) DriverBinding ComponentName2 ComponentName
	USB mouse driver	UsbMouseDxe	Consumes the USB I/O Protocol and produces the Simple Pointer Protocol. 28: Image(UsbMouseDxe) DriverBinding ComponentName2 ComponentName

Graphics	Graphics Output	CirrusLogic5430 Dxe	Consumes the PCI I/O Protocol and produces the Graphics Output Protocol. 2E: Image(CirrusLogic5430Dxe) DriverBinding ComponentName2 ComponentName
	Graphics console driver	GraphicsConsole Dxe	Consumes the Graphics Output Protocol and produces the Simple Text Output Protocol. 2D: Image(GraphicsConsoleDxe) DriverBinding ComponentName2 ComponentName
Serial	PCI Serial 16550 UART driver	PciSerialDxe	Consumes the PCI I/O Protocol and produces the Serial I/O Protocol. 30: Image(PciSerialDxe) DriverBinding ComponentName2 ComponentName
	Serial terminal driver	TerminalDxe	Consumes the Serial I/O Protocol and produces the Simple Text Input, Simple text Input Ex, and Simple Text Output Protocols. 31: Image(TerminalDxe) DriverBinding ComponentName2 ComponentName
Generic Console	Platform console management driver	ConPlatformDxe	This driver is unique in that a single set of driver code produces two driver handles—one for the “Console Out” and another for the “Console In”. This driver evaluates the set of physical console devices and the UEFI Console Variables that describe the platform settings for active consoles and marks the active consoles so they can be easily discovered by ConSplitterDxe. Different platforms may modify the default policy decisions this driver provides. 32: Image(ConPlatformDxe) Driver Binding ComponentName2 ComponentName 33: DriverBinding ComponentName2 ComponentName

	Console splitter driver	ConSplitterDxe	<p>This driver may not be present on all platforms. It is only required on platforms that support multiple output console devices or multiple input console devices. It combines the various selected input and output devices for the following four basic UEFI user devices:</p> <p>ConIn ConOut ErrOut PointerIn</p> <p>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:</p> <p>PrimaryConIn PrimaryConOut PrimaryStdErr</p> <p>The virtual handles always exist even if no console exists or no consoles are yet connected in the system.</p> <pre> 34: Image(ConSplitterDxe) DriverBinding ComponentName2 ComponentName 35: DriverBinding ComponentName2 ComponentName 36: DriverBinding ComponentName2 ComponentName 37: DriverBinding ComponentName2 ComponentName 38: TxtinEx Txtin SimplePointer AbsolutePointer 39: Txtout GraphicsOutput UgaDraw </pre>
--	-------------------------	----------------	--

3.15.5 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 are stored in the *ConIn*, *ConOut*, and *ErrOut* global UEFI variables (see the Boot Manager chapter of the *UEFI 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 the following:

- The *ErrOut* and *ConOut* variables are multi-instance device paths separated by semicolon (;) indicating that the EFI output is mirrored on two different console devices. The mirroring is performed when the *ConSplitterDxe* driver is connected. In this example, the two devices are a serial terminal and a PCI video controller.
- The *ConIn* variable contains a device path to a serial terminal.
- The *ErrOut* variable is typically the same as the *ConOut* variable, but could be redirected to different set of devices. It is important to check how this UEFI variable is configured when developing UEFI drivers because the debug messages from a UEFI Driver are typically directed to the console device(s) specified by *ErrOut*.*ErrOut* may not specify the same devices as *ConOut*

In this example, the two devices are a serial terminal and a PCI video controller. The EDK II provides the *DebugLib* which is a library that provides services such as *DEBUG()* and *ASSERT()* that are used generate debug messages.

3.15.6 ConIn

The platform connects the console devices using the device paths from the *ConIn*, *ConOut*, and *ErrOut* global UEFI variables. The *ConIn* connection process is discussed first.

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

The UEFI connection process searches for the device in the handle database having a device path that most closely matches the following.

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

It finds 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))
```

UEFI calls `ConnectController()`, passing in handle 17 and the remaining device path. The connection code constructs a list of all the drivers in the system and calls each driver, passing handle 17 and the remaining device path into the `Supported()` service. The only driver installed in the handle database that returns `EFI_SUCCESS` for this device handle is handle 30:

```
30: Image(PciSerialDxe) DriverBinding ComponentName2 ComponentName
```

After `ConnectController()` finds a driver that supports handle 17, it passes device handle 17 and the remaining device path `Uart(9600 N81)/ VenMsg(Vt100+)` into the serial driver's `start()` service. The serial driver opens the PCI I/O Protocol on handle 17 and create a new child handle. The following is installed onto the new child handle:

- `EFI_SERIAL_IO_PROTOCOL` (defined in the Console Support chapter of the *UEFI 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 is 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 finds the terminal driver, returning `EFI_SUCCESS` from the `Supported()` service.

```
31: Image(TerminalDxe) DriverBinding ComponentName2 ComponentName
```

This driver's `Start()` service opens the `EFI_SERIAL_IO_PROTOCOL`, creates a new child handle, and installs the following:

- `EFI_SIMPLE_TEXT_INPUT_PROTOCOL`
- `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL`
- `EFI_DEVICE_PATH_PROTOCOL`

The console protocols are defined in the Console Support chapter of the *UEFI Specification*. The device path is 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, shown below, is the new child handle.

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

At this point, the process still has not completely matched the `ConIn` device path, so the connection process is 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. The search for a driver that supports this controller results in two driver handles that return `EFI_SUCCESS` to the `Supported()` service. The two driver handles are from the platform console management driver:

```
32: Image(ConPlatformDxe) Driver Binding ComponentName2 ComponentName
33: Driver Binding ComponentName2 ComponentName
```

Driver 32 installs a `ConOut` Tag 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 also installs a `StdErr` Tag GUID on the handle if the device path is listed in the `ErrOut` global UEFI variable. This case is also true in the following example. Therefore, handle 3C has two new protocols on it: `ConOut` and `StdErr`.

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

Driver 33 installs a `ConIn` Tag GUID on the handle if the device path is listed in the `ConIn` global UEFI variable (which it does because the connection process started that way), so handle 3C has the `ConIn` protocol attached.

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

UEFI uses these three protocols (`ConIn`, `ConOut`, and `StdErr`) to mark devices in the platform, which have been selected by the user as `ConIn`, `ConOut`, and `StdErr`. These protocols are actually Tag GUIDs without any services or data.

There are three other driver handles that return `EFI_SUCCESS` from the `Supported()` service. These driver handles are from the console splitter drivers for the `ConIn`, `ConOut`, and `StdErr` 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(ConSplitterDxe) DriverBinding ComponentName3 ComponentName
35: DriverBinding ComponentName2 ComponentName
36: DriverBinding ComponentName2 ComponentName
37: DriverBinding ComponentName2 ComponentName
```

Remember that when the console splitter driver was first loaded, it created three virtual handles for the primary console input device, the primary console output device,

and the primary standard error device.

```
38: TxtinEx TxtIn SimplePointer AbsolutePointer  
39: Txtout GraphicsOutput UgaDraw  
3A: Txtout
```

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 then opens the `ConIn` protocol on handle 3C such that the primary console input device handle 38 becomes a child controller and starts aggregating the `SIMPLE_INPUT_EX_PROTOCOL` and `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 primary console output handle 39.

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 primary standard error handle 3A.

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 have been installed.

3.15.7 ConOut

As with `ConIn`, firmware connects the `ConOut` devices using the device paths in the `ConOut` global UEFI variable. If `ConIn` was not complicated enough, the `ConOut` global UEFI 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 searches 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  
(Acpi(HWP0002,0,PNP0A03)/Pci(1|1)/Uart(9600  
N81)/VenMsg(Vt100+))
```

UEFI performs a `ConnectController()` on handle 3C. Because this step was previously done 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 searches 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))
```

UEFI now performs 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 returns `EFI_SUCCESS` is the GraphicsOutput driver.

```
2E: Image(CirrusLogic5430Dxe) Driver Binding ComponentName2  
ComponentName
```

`ConnectController()` calls this driver's `Start()` function and `Start()` consumes the device's `EFI_PCI_IO_PROTOCOL` and installs the `EFI_GRAPHICS_OUTPUT_PROTOCOL` onto the device handle 1C.

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

`ConnectController()` continues to process its list of drivers and finds that the "GraphicsConsole" driver's `Supported()` service returns `EFI_SUCCESS`.

```
2D: Image(GraphicsConsoleDxe) DriverBinding ComponentName2  
ComponentName
```

Next, the graphics console driver's `Start()` service consumes the `EFI_GRAPHICS_OUTPUT_PROTOCOL` and produces 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()` continues to process its list of drivers, now searching for a driver that supports this controller, and finds two driver handles that return `EFI_SUCCESS` from their `Supported()` services. These two driver handles are from the platform console management driver:

```
32: Image(ConPlatformDxe) DriverBinding ComponentName2 ComponentName
```

Driver handle 32 installs a `ConOut` Tag GUID on the handle if the device path is listed in the `ConOut` global UEFI variable. In this example, the case is true. Driver 32 also installs a `StdErr` Tag 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 `StdErr`.

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

These two protocols (`ConOut` and `StdErr`) are used to mark devices in the system that have been user-selected as `ConOut` and `StdErr`. These protocols are actually just Tag GUID without any functions or data.

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

```
36: DriverBindingComponentName  
37: DriverBindingComponentName
```

Remember that when the console splitter driver was first loaded, it created three virtual handles.

```
38: TxtinEx TxtIn SimplePointer AbsolutePointer  
39: Txtout GraphicsOutput UgaDraw  
3A: Txtout
```

The console splitter driver's `Supported()` service for driver handle 36 examines the handle 1C for a `ConOut` Protocol. Having found it, `EFI_SUCCESS` is returned. The `Start()` service then opens the `ConOut` protocol on *device handle* 1C such that the *device handle* 39 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 *device handle* 3A.

3.15.8 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)
```

3.15.9 Boot Manager Connect All Processing

On some platforms, the 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 platform follows the following sequence:

1. 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 have a Device Path Protocol, and no parent controllers.
2. `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.

3. As each bus is expanded, and if the bus supports storage devices for UEFI drivers, additional UEFI drivers are then loaded from those storage devices (for example, option ROMs on PCI adapters).
4. 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 are produced.
5. When the process is complete, the entire tree of boot devices in the system hierarchy is present in the handle database.

3.15.10 Boot Manager Driver List Processing

The platform boot manager loads the drivers that are specified by the `DriverOrder` and `Driver####` environment variables. These environment variables are discussed in more detail in the Boot Manager chapter of the *UEFI Specification*.

Before the platform boot manager loads each driver, it uses 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 platform boot manager uses 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 lower 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 is still managing the same device.

However, if the newer driver in the `DriverOrder` list has a load attribute of `LOAD_OPTION_FORCE_RECONNECT`, then the platform boot manager disconnects and reconnects all the controllers, so the driver with the highest version number manages the same device that the lower versioned driver used to manage. 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 increases the boot time.

3.15.11 Boot Manager BootNext Processing

After connecting any drivers in the `DriverOrder` list, the platform boot manager attempts to boot the option that is specified by the `BootNext` environment variable. This environment variable is discussed in the Boot Manager chapter of the *UEFI Specification*. This variable typically is not set, but if it is, the platform firmware deletes the variable and then attempts to load the boot option that is described in the `Boot####` variable pointed to by `BootNext`.

Before the platform boot manager boots the boot option, it uses 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`.

3.15.12 Boot Manager Boot Option Processing

The platform boot manager displays the boot option menu and if the auto-boot `TimeOut` environment variable has been set, then the first boot option is loaded when the timer expires. The boot options can be enumerated by the platform boot manager by reading the `BootOrder` and `Boot####` environment variables. These environment variables are more thoroughly discussed in the Boot Manager chapter of the *UEFI Specification*. A boot option is typically an OS loader that never returns to UEFI, but boot options can also be UEFI applications like diagnostic utilities or the UEFI Shell.

If a boot option does return to the platform boot manager, and the return status is not `EFI_SUCCESS`, then the platform boot manager processes the next boot option. This process is repeated until an OS is booted, `EFI_SUCCESS` is returned by a boot option or the list of boot options is exhausted. Once the boot process has halted, the platform boot manager may provide a user interface that allows the user to manually boot an OS or manage the platform.

The platform boot manager uses 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`.

4

General Driver Design Guidelines

This chapter contains general guidelines for the implementation of all types of UEFI drivers. Guidelines for specific driver types (PCI, USB, SCSI, ATA, Console, Graphics, Mass Storage, Network, etc.) are presented in individual chapters later in this guide. This chapter also focuses on general guidelines for implementing UEFI Drivers sources portable to all UEFI conformant platforms and all CPU architectures supported by the *UEFI Specification*. If these guidelines are followed, there is a good chance that UEFI Drivers can be re-compiled for a different CPU architecture with no source code changes.

There are a few portability issues that apply specifically to IPF and EBC, and these are presented in individual sections later in this guide as well. The summary of topics covered includes:

- Common practices for C source code
- Maximizing Platform Compatibility
- Maximizing CPU Compatibility
- Optimizing for size and performance

4.1 Common Coding Practices

This section covers common coding practices for implementing UEFI Drivers. Following these practices may improve a UEFI Driver's compatibility with different C compilers. The most important rule to follow is to use ANSI C and to avoid the use of compiler specific language extensions. *Avoiding the use of assembly language is also recommended.*

A common approach when implementing a new UEFI Driver is to find an existing UEFI Driver with similar features and functionality and use that existing UEFI Driver as a starting point for the new UEFI Driver. [Appendix B](#) contains a table that lists some example UEFI Drivers provided in the EDK II and the features implemented by those UEFI Drivers. The EDK II contains many more UEFI drivers than those listed in [Appendix B](#).

4.1.1 Type Checking

Some compilers perform stronger type checking than other compilers such as the Intel family of compilers including the Intel® C Compiler for EFI Byte Code. As a result, code that compiles without any errors or warnings on one compiler may generate warnings or errors when compiled with another compiler. The following example shows two common examples from UEFI Drivers that use `AllocatePool()` and `OpenProtocol()`. These examples show the style that may generate warnings with some compilers, and the correct method to prevent the warnings.

```
#include <Uefi.h>
#include <Protocol/BlockIo.h>
#include <Protocol/DriverBinding.h>
#include <Library/UefiBootServicesTableLib.h>

typedef struct {
    UINT8 First;
    UINT32 Second;
} MY_STRUCTURE;

EFI_STATUS Status;
EFI_DRIVER_BINDING_PROTOCOL *This;
EFI_HANDLE ControllerHandle;
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 7—Stronger type checking

4.1.2 Avoid Name Collisions

Compilers and linkers guarantee that there are no function name or global variable name collisions within a single UEFI Driver, but the compilers and linkers cannot check for function name or global variable name collisions between UEFI Drivers. This inability to check is a concern when debuggers are used that can perform source-level debugging or can display function names. [Appendix A](#) contains source code templates that help avoid function name collisions between UEFI Drivers by using the name of the driver in the function names.

4.1.3 Maximize Warning Levels

To catch possible issues with assigning or comparing values of different sizes, UEFI drivers should always be compiled with the highest warning level possible. For example, the Microsoft™ compilers support the /WX and /W3 or /W4 compiler flags. The /WX flag causes any compile time warnings to generate an error, so the build stops 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 generate a warning. With the /WX flag, the compile stops when such size mismatches are detected.

If a UEFI Driver is being developed for a 32-bit architecture and is expected to be ported to a 64-bit architecture, it is a good idea to compile the UEFI driver with a 64-bit compiler during the development process. This helps ensure the code is clean when validation on the 64-bit processor is begun. By using the /WX and /W3 or /W4 compiler flags, any size mismatches that are generated by only 64-bit code are detected.

TIP: As the warning levels are increased, a compiler may produce more error messages. This helps develop more robust, portable code.

4.1.4 Compiler Optimizations

Test UEFI Drivers built with compiler optimizations enabled and disabled. This helps identify odd code errors that might not manifest at lower optimization levels. It also helps identify if a UEFI Driver is sensitive to differences in execution speed. A UEFI Driver that works at lower levels of optimization, but not at higher levels, may be missing logic for a required synchronization.

4.2 Maximize Platform Compatibility

UEFI drivers should make as few assumptions about a system's architecture as possible. Minimizing the number of assumptions maximizes the UEFI driver's platform compatibility. It also reduces the amount of driver maintenance that is required when a UEFI Driver is deployed on new platforms.

4.2.1 Never Assume all UEFI Drivers are Executed

Typically, the same vendor that produces a UEFI driver also produces 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 it only needs to install the driver on one SCSI bus in order 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 a UEFI driver for a short period of time and released, or it may have been managed by a 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.

Note: *OS drivers shall not make assumptions that the UEFI driver has initialized or configured the device in any way.*

Note: *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.*

4.2.2 Eliminate System Memory Assumptions

Do not make 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 in any system and entire memory regions may be missing. For example, some memory regions could already be allocated (such as for an I/O device), some memory may be non-addressable, and/or physical memory could actually be missing. UEFI is designed for a wide variety of platforms. As such, portable drivers should not have hard-coded limits. Instead, they should rely on published specifications, UEFI, 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 Specification*. For example, memory is not always 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 your drivers should not expect them to be present.

4.2.3 Use UEFI Memory Allocation Services

The **AllocatePool()** service *does not* allow the caller to specify a preferred address so this service is always safe to use and has no impact on platform compatibility. The **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 increases platform compatibility. The allocation types of **AllocateMaxAddress** and **AllocateAddress** may reduce platform compatibility. Refer to [Chapter 5](#) in this guide for information about using the **AllocatePages** service.

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 a UEFI driver allocates and the DMA buffer addresses that DMA bus master's use. A UEFI driver should not allocate buffers from specific addresses or below specific addresses. These types of allocations may fail on different system configurations. The following rules help ensure a UEFI Driver makes appropriate memory allocations.

- Use natural alignment (byte values on byte boundaries) when allocating buffers. This maximizes portability and helps avoid alignment faults on IPF platforms.
- Buffers allocated on a 32-bit CPU architecture using the UEFI Boot Service `AllocatePool()` are guaranteed to be below 4GB.
- Buffers allocated on a 64-bit CPU architecture using the UEFI Boot Service `AllocatePool()` may be above 4GB if memory is present above 4 GB.
- The UEFI Boot Service `AllocatePages()` may be used to allocate a buffer anywhere system memory is present. This means `AllocatePages()` may return a buffer on a 32-bit CPU architecture that is above 4 GB if memory is present above 4GB and that buffer can never be accessed.
- 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.
- To prevent memory leaks, every allocation operation must have a corresponding free operation.
- Test UEFI drivers on 64-bit architectures with memory configurations where system memory is present above 4GB.
- Test UEFI drivers on 64-bit architectures with memory configurations where system memory is not present above 4GB.
- UEFI drivers should not use fixed-size arrays. Instead, memory resources should be dynamically allocated using the `AllocatePages()` and `AllocatePool()` services.

4.2.4 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 are always a fixed number of controllers. This design maximizes 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](#) of this guide introduces the private context data structure, which is a lightweight mechanism that allows a UEFI driver to be designed with no limitations on the number of controllers that the UEFI driver can manage.

4.2.5 Never Directly Access Hardware Resources

A UEFI driver should also never directly access any system chipset resources. Directly accessing these resources limits the compatibility of the UEFI driver to systems only with that specific chipset.

Instead, the UEFI boot services, UEFI runtime services, and various protocol services should be used to access the system resources that are required by a UEFI driver. The driver should look only for services to hook into—a capability, such as a PCI bus. The driver then consumes the protocols necessary for accessing that capability. It is the chipset's responsibility to get that capability ready for use.

TIP: The general rule is to only access the hardware that the UEFI Driver supports and use abstractions for other hardware. If there is not an abstraction for a system device, do not use the device. That device may change interface and functionality in the future.

This recommendation serves several purposes. By using the software abstractions provided by the platform vendor, the UEFI driver maximizes 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 29](#) in this guide 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.

Putting effort into source code portability helps maximize future platform compatibility.

4.2.6

Memory ordering

Not all processors have strongly ordered memory models and some compilers, when high levels of optimization are enabled, may induce memory ordering issues. Weak ordering means that the order in which memory transactions are presented in the C source code may not be the same order of operations when the code is executed. IPF platforms are weakly ordered, so UEFI Drivers that are compiled for IPF have to be aware of this issue. See *A Formal Specification of Intel Itanium Processor Family Memory Ordering* for a detailed discussion of this topic. It is also discussed in the *Intel Itanium Architecture Software Developer Manuals*.

TIP: Most of the details of memory ordering are taken care of by protocols and libraries. If protocols and libraries are used to access hardware, then memory ordering issues should be hidden from a UEFI Driver implementation. A direct access to hardware is **not recommended**.

Normally, memory ordering is not an issue, because the processor and the compiler guarantee that the code executes as the developer expects. However, UEFI drivers that access 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 resolve its own memory ordering issues, but a UEFI Driver is responsible for managing the processor's ordering issues.

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 C data structures or portions of C data structures with the `volatile` attribute. The compiler ensures that strong ordering is used for all operations to with that declaration.
- **Technique 2:** Use the EDK II library `BaseLib` function called `MemoryFence()`. This function guarantees that all the transactions in the source code prior to the `MemoryFence()` function are completed before the code after the `MemoryFence()` function is executed. On IPF platforms, this function executes a memory fence instruction. Some compilers provide an intrinsic function that declares a barrier and, if this intrinsic is provided, the EDK II implementation of `MemoryFence()` includes that barrier intrinsic. The barrier intrinsic is not really a call. Instead, it prevents memory read/write transactions from being moved across the barrier as part of the compiler code generation. This may be very important when high levels of compiler optimization are enabled.

The second technique is 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 (it is off in a structure definition). In addition, the `volatile` declaration may impact the performance of the UEFI Driver’s because all memory transactions to the structure are strongly ordered.

It is recommended that these techniques be used appropriately in all driver types to maximize the UEFI driver’s platform compatibility.

4.2.7 DMA

System memory buffers used for DMA should not be allocated from a specific address or below a specific address. In addition, UEFI drivers must always use I/O abstractions to setup and complete DMA transactions.

It is not legal to program a system memory address into a DMA bus master. This programming works on chipsets that have a one-to-one mapping between system memory addresses and PCI DMA addresses, but it does not work with chipsets that remap DMA transactions.

4.2.8 Supporting Older EFI Specifications and UEFI Specifications

Complying with different versions of the *EFI Specification* and *UEFI Specification* may be critical for some UEFI Driver implementations. If the driver is required to work on platforms that are conformant with the older *EFI Specifications* or *UEFI Specifications* and also on current and next-generation UEFI systems, then the UEFI Driver design must consider the requirements from multiple EFI/UEFI Specifications.

In many cases, the UEFI Driver can produce extra protocols to increase compatibility. In other cases, the UEFI Driver may be required to detect the UEFI capabilities provided by the platform firmware and adjust the protocols that the UEFI Driver consumes and produces.

4.2.9 Reduce Poll Frequency

UEFI drivers operate in a polled mode and do not use interrupts. For example, 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.

A common mistake in UEFI drivers is polling too often.

Remember that polling, versus interrupts, is a pull model, not a push model. The trade-off in a polling system is how fast the device is polled (which can degrade system performance) versus how responsive the driver is to that request. For example, in a polling system, the driver should not send a request to a device and wait until that device responds before moving on to another task. In general, the polling 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 a UEFI-enabled platform degrades if too many UEFI drivers create high-frequency periodic timer events.

Note: *It is recommended that the period of a periodic timer event be at least 10 ms. In general, the period should be as large as possible based upon a specific device's timing requirements. Most drivers can use events with timer periods in the range of 100 ms to several seconds.*

When initially writing the driver, an estimate can be made for the initial polling frequency. However, the polling frequency may have to be adjusted based on an analysis of the driver's performance on an actual machine.

TIP: As part of the development process, make sure time is reserved for performance analysis to find out how much time is taken up polling each device.

4.2.9.1 Distinguishing a polling issue versus another type of bug

The symptoms of a polling issue versus some other type of bug can look nearly identical. The key to identifying a polling issue is: Don't assume anything. Begin simply by performing an analysis to get data—the time taken by each task can be measured. If a task is taking longer than expected, the code associated with that task can then be examined more closely.

4.2.10 Minimize Time in Notification Functions

UEFI drivers should not spend a lot of time in their event notification functions because this blocks the normal execution mode of the system. A 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 II that uses periodic timer events.

4.2.11 Use Proper Task Priority Levels

The TPLs provide a mechanism for code to run at a higher priority than application code. One can be running the UEFI Shell, and a 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 events are not used. They may still use the **RaiseTPL()** and **RestoreTPL()** for critical sections.

Driver diagnostics are typically just applications. They do not normally need to use TPLs or events unless the diagnostics is testing the TPL or event mechanisms in EFI. However, 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, it should be done for the shortest possible time interval.

Caution: *There are ways in which the platform firmware can be put into an undefined state by misuse of the **RaiseTPL()** and **RestoreTPL()** functions.*

Caution: *Do not misuse the **RaiseTPL()** service by raising the task priority level too high for an extended period of time. Raising the TPL level above **TPL_APPLICATION** circumvents the timer tick. This can interfere with other drivers, applications, and other elements that rely on the timer tick. It can cause extreme, and sometimes catastrophic slowing of the system. It can cause other drivers, applications, and other things that rely on the timer tick to fail. Always mirror the raise TPL service with the restore TPL service.*

4.2.12 Design to be re-entrant

Design all UEFI Drivers to manage multiple controllers. This requires that the controller specific information be managed in its own data structure. 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, collect data into a private context data structure so that each time an I/O protocol installs onto a handle, a new version of the structure is allocated from memory. This concept is described in detail in [Chapter 8](#) of this guide.

4.2.13 Do not use hidden PCI 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 the PCI bus driver's responsibility to extract the option ROM contents when a PCI bus enumerates. If code were required to access hidden portions of an option ROM, then the PCI bus driver would not have the ability to extract the additional PCI Option ROM contents.

This inability means that the UEFI drivers in a PCI Option ROM must be visible without accessing a hidden portion of a PCI Option ROM. However, if there is a safe mechanism to access the hidden portions of the PCI option ROM after the UEFI drivers have been loaded and executed, then the UEFI driver may choose to access those contents. For example, non-volatile configuration information, utilities, or diagnostics can be stored in the hidden PCI Option ROM regions.

Caution: *The hidden option ROM regions are also not measurable via UEFI 2.3 and beyond signing and verification interfaces. This makes them, and the system, less secure.*

4.2.14 Store Configuration Data with Device

The configuration for a UEFI driver should be stored on the same field replaceable unit (FRU) as the managed device. If a UEFI driver is stored on the motherboard, then the driver's configuration information can be stored in UEFI variables. If a UEFI driver is stored in an add-in card, then the driver's configuration information should be stored in the NVRAM provided on the add-in card.

4.2.14.1 Benefits

This method ensures that it is possible to statically determine the maximum configuration storage that is required for the FRU during FRU design. In particular, if option cards stored their configuration in UEFI variables, the amount of variable storage could not be statically calculated because it generally is not possible to know the particular set of option cards installed in a system ahead of time. 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 configuration data in the same FRU as the device reduces the amount of stale data left in UEFI variables. If an option card stored its data in UEFI variables and was then removed, there would be no automatic cleanup mechanism to purge the UEFI variables associated with that card.

Storing configuration data in the same FRU as the device 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. It can also maintain preconfigured spares.

4.2.15 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 PNPOA03 identifier in the `_HID` to identify the PCI root bridge. The following are the only requirements for the PCI root bridge:

- The PNPOA03 identifier must appear in `_HID` if a vendor-specific capability description isn't needed.
- The PNPOA03 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.

4.2.15.1 PNPID byte order for UEFI

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
```

The significance is that operating systems that try to match the UEFI ACPI device path node to the ACPI name space must perform a translation.

Refer to [Chapter 4](#) of this guide for information about lengths of words on 32-bit versus 64-bit architectures.

4.2.15.2 Working with UEFI Device Path Nodes

UEFI Device Paths Nodes are not required to be aligned. If the proper coding style is used when working with device paths, a UEFI Driver can be implemented to guarantee all that fields of UEFI Device Path Nodes are accessed with natural alignment. This improves platform compatibility, especially for IPF platforms.

TIP: Do not assume that, when given a device path, that the path is aligned. Copy pieces of the device path to a known-aligned device path before accessing it. The device path may then be accessed safely. Alternatively, use EDK II **BaseLib** functions to perform unaligned reads and writes.

4.2.16 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.

Caution: *On a boot or reboot, there shall be no writes to shared storage without user acknowledgement. Any writes to shared storage by a UEFI driver may corrupt shared storage as viewed by another system. As a result, all outstanding I/O in the controller's buffers will be cleared, as well as any internal. Any I/O operations that occur after a reboot may corrupt shared storage.*

Caution: *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.*

Caution: *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.*

Caution: *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.*

4.2.17 Limit use of Console Services

PC BIOS legacy option ROMs typically display banners and allow hotkey(s) to enter the configuration area for a particular card. Current UEFI drivers use HII functionality to allow access to system configuration areas.

Because UEFI drivers now have HII functionality, 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 typically works if the driver is loaded after the UEFI console is connected. However, because console drivers may live in option ROMs, some firmware implementations may load the option ROM drivers before the UEFI console is connected. In such cases, the `ConOut` and `StdErr` fields of the UEFI system table may be `NULL`, and printing can crash the system. The `DEBUG()` macro should check to see if the field is `NULL` before using those services.

4.2.18 Offer alternatives to function keys

Configuration of drivers should be accomplished via HII and via OS-present interfaces.

There are design considerations when interacting outside of configuration. First, consider using the setup interface as the user interface for a UEFI driver. The user already understands the interface and remote use is already enabled. If the existing high level interfaces cannot be used, then follow the design considerations for using console based services.

UEFI drivers should use the console input services (see [Section 22.2](#) of this guide), and then be aware of alternatives to function keys. This is because the UEFI console may be connected through a serial port. In such cases, it is sensitive to the correct terminal emulator configuration. If the terminal emulator is not correctly configured to match the terminal settings in UEFI (PC ANSI, VT100, VT100+, or VT-UTF8), some of the keys (function keys, arrow keys (page up/down, insert/delete, and backspace), may not work correctly, display colors properly nor render the correct cursor positioning.

Note: *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.*

Note: *It is important to be aware 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 14](#), following, 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 decides if alternate key sequences are supported and which alternate mappings should be used. The table also lists the UEFI scan code from the Simple Input Protocol and the alternate key sequence to use to produce particular scan codes.

Most of these key sequences are directly supported in the EDK II—special handling is not required to support these key sequences on a remote terminal. Those labeled as “No” are not directly supported in the EDK II. They are parsed and interpreted by the configuration protocol or application.

Table 14—Alternate key sequences for remote terminals

UEFI scan code	Key sequence	Supported in EDK II?
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

4.3

Maximize CPU Compatibility

UEFI Drivers should be designed to maximize source code portability since it is possible to write a single UEFI Driver that compiles on all CPU architectures supported by the *UEFI Specification*. The list of supported CPU architectures may grow over time, so it is important to follow these portability guidelines.

The guidelines presented here apply to all CPU architectures. [Chapter 28](#) covers portability issues specific to IPF platforms, and [Chapter 29](#) covers portability issues that are specific to EBC.

When porting between CPU architectures, most developers take as much existing code as possible and reuse it. Unfortunately, some developers porting code do not rigorously follow the UEFI conventions, such as using only the data types defined in the Calling Conventions section of the *UEFI Specification*. Others may not follow best coding practices.

- Use data types defined by the Calling Conventions section of the *UEFI Specification*.
- Use compiler flag settings to guarantee that the UEFI calling conventions for the CPU architecture are followed. See the Calling Conventions section of the *UEFI Specification* for details.
- If a UEFI driver contains assembly language sources, then either the source needs to be ported or it needs to be converted to C language source. Conversion to C language source is **recommended**. The EDK II library **BaseLib**, and other EDK II libraries, provide functions that may reduce, or even eliminate, the need to assembly code in UEFI Drivers.

TIP: Implement UEFI Drivers in C to maximize portability,

- Avoid use of C++. It is not supported by EBC.
- Avoid unaligned data accesses. Compilers, by default, generate code and data that perform aligned accesses. Unaligned data accessed are generated when features such as byte-packed structures, type casting pointers, or assembly language are used. Aligned data accesses typically execute faster than unaligned data accesses. Parsing UEFI Device Paths is a common generator of unaligned data accesses. These generate alignment faults on IPF platforms.
- The best approach to debugging a UEFI Driver ported to a differing CPU architecture is to keep a good code base with every revision. This allows comparison with earlier revisions to see the source code before and after the problem became visible.
- If source code is not available, the CPU register state may not be sufficient to debug a specific issue. Keep in mind that a "new" problem might have nothing to do with a recent change to the code. A pre-existing problem might not have shown up before for a variety of reasons. For example, the current developer might have included error checking or exercised the error handling registers after making an addition to the code—error checking that might not have been done before. Or a new addition might make the pre-existing problem worse, so the problem finally becomes visible in the new revision.
- Perform a minimal port first to test simple parts of the UEFI driver. This is simply good porting practices, but even experienced developers can forget to port and test the simple things first. Start with a known-good sample driver that is extremely simple. For example, a driver that prints "Hello World". Then divide the code into sections. Begin inserting and testing the less complicated sections into the known-good driver, one section at a time. Another technique is to replace more complex code with "neutered" code that returns but doesn't actually do anything. Make sure the simple sections work and do not cause alignment faults or other errors. Only then should the more complicated sections be added and adapted to the new architecture rules. This approach can significantly cut down on debug time.

4.3.1 Assignment and comparison operators

There are issues that, if a data value is cast from a larger size to a smaller size, the upper bits of the larger values are stripped. In general, this stripping causes a compiler warning, so these are easy issues to catch. However, there are a few cases where compilation is free of errors and warnings on 32-bit platforms but generates errors or warnings on 64-bit platforms. The only way to guarantee catching these errors early on

is to compile for both 32-bit and 64-bit processors during the entire development process.

When a warning is generated by a 64-bit processor, it 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 are stripped.

The example below shows several examples that generate a warning and how to eliminate it with an explicit cast. The last example is the most interesting because it does not generate any warnings on a 32-bit architecture, but does on 64-bit. This difference is because a **UINTN** on 32-bit CPUs is identical to **UINT32**, but **UINTN** on 64-bit CPUs is identical to a **UINT64**.

```
#include <Uefi.h>

UINT8    Value8;
UINT16   Value16;
UINT32   Value32;
UINT64   Value64;
UINTN    ValueN;

//
// Warning generated on 32-bit CPU
// Warning generated on 64-bit CPU
//
Value8 = Value16;

//
// Works, upper 8 bits stripped
//
Value8 = (UINT8)Value16;

//
// Works
//
Value16 = Value8;

//
// Warning generated on 32-bit CPU
// Warning generated on 64-bit CPU
//
Value8 = Value32;

//
// Works, upper 24 bits stripped
//
Value8 = (UINT8)Value32;

//
// Works
//
Value32 = Value8;

//
// Warning generated on 32-bit CPU
// Warning generated on 64-bit CPU
//
Value8 = Value64;

//
// Works, upper 56 bits stripped
//
Value8 = (UINT8)Value64;
```

```

//  

// Works  

//  

Value64 = Value8;  

//  

// Warning generated on 32-bit CPU  

// Warning generated on 64-bit CPU  

//  

Value8 = ValueN;  

//  

// Works  

// Upper 24 bits stripped on 32-bit CPU  

// Upper 56 bits stripped on 64-bit CPU  

//  

Value8 = (UINT8)ValueN;  

//  

// Works  

//  

ValueN = Value8;  

//  

// Works on 32-bit CPU  

// Warning generated in 64-bit CPU  

//  

Value32 = ValueN;  

//  

// Works on 32-bit CPU  

// Upper 32-bits stripped on 64-bit CPU  

//  

Value32 = (UINT32)ValueN;

```

Example 8—Assignment operation warnings

Example 9, below, is very similar to Example 8 except the assignments have been replaced with comparison operations. The same issues shown are 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 32-bit platforms with no errors or warnings but generate warnings on 64-bit architectures. 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 strips the upper bits of the operand. When a cast is performed to `INTN` or `UINTN`, a different number of bits are stripped for 32-bit and 64-bit architectures.

```

#include <Uefi.h>

UINT64  ValueU64;
UINTN   ValueUN;
INT64   Value64;
INTN    ValueN;

//  

// Works on 32-bit CPU  

// Warning generated in 64-bit CPU  

//  

if (ValueU64 == ValueN) {}

```

```

//  

// Works on 32-bit CPU  

// Warning generated in 64-bit CPU  

//  

if (ValueUN == Value64) {}  

//  

// Works on 32-bit CPU  

// Warning generated in 64-bit CPU  

//  

if (Value64 == ValueUN) {}  

//  

// Works on 32-bit CPU  

// Warning generated in 64-bit CPU  

//  

if (ValueN == ValueU64) {}  

//  

// Works on 32-bit and 64-bit CPUs  

//  

if ((INTN)ValueU64 == ValueN) {}  

//  

// Works on 32-bit and 64-bit CPUs  

//  

if ((INT64)ValueUN == Value64) {}  

//  

// Works on 32-bit and 64-bit CPUs  

//  

if ((UINTN)Value64 == ValueUN) {}  

//  

// Works on 32-bit and 64-bit CPUs  

//  

if ((UINT64)ValueN == ValueU64) {}  

//  

// Works on 32-bit and 64-bit CPUs  

//  

if (ValueU64 == (UINT64)ValueN) {}  

//  

// Works on 32-bit and 64-bit CPUs  

//  

if (ValueUN == (UINTN)Value64) {}  

//  

// Works on 32-bit and 64-bit CPUs  

//  

if (Value64 == (INT64)ValueUN) {}  

//  

// Works on 32-bit and 64-bit CPUs  

//  

if (ValueN == (INTN)ValueU64) {}

```

Example 9—Comparison operation warnings

4.3.2 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 depending on the platform architecture, such as 32-bit versus 64-bit platforms. 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 may not run on 64-bit platforms with physical memory above 4 GB.

4.3.3 Converting pointers

Be mindful when converting physical addresses to pointers on 64-bit architectures. All UEFI driver writers must be aware that pointers may contain values above 4 GB, and that care must be taken never to strip the upper address bits. If the upper address bits are stripped, the driver *may* work on 32-bit architectures, and on 64-bit architectures with small memory configurations, but *may not* work on 64-bit platforms with larger memory configurations.

Note: *Make sure the driver does not strip the upper address bits when converting pointers.*

4.3.3.1 The Exception to the Rule

There is one exception to this rule of casting pointers and it applies to both 32-bit and 64-bit processors. The data types **INTN** and **UINTN** are the exact same size of pointers on both 32-bit and 64-bit 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. Also, 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.

4.3.3.2 Identifying a Pointer Problem

Problems caused by mistakes in pointer casting are difficult to catch. This is so 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 are generated. In fact, the code may execute fine on, for example, 32-bit platforms and on 64-bit platforms with physical memory below 4 GB. The only failing case is when the code is tested on a 64-bit system with physical memory above 4 GB. The symptom is typically a processor exception that results in a system hang or reset.

The example 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.

```
#include <Uefi.h>

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 32-bit CPU.
                                         // Strips upper 56 bits on 64-bit CPU.
ValueU16 = (UINT16)MyStructure;          // Bad. Strips upper 16 bits on 32-bit CPU.
                                         // Strips upper 48 bits on 64-bit CPU.
ValueU32 = (UINT32)MyStructure;          // Bad. Works on 32-bit CPUs.
                                         // Strips upper 32 bits on 64-bit CPU.
ValueU64 = (UINT64)MyStructure;          // Good. Works on all architectures
Value64 = (INT64)MyStructure;            // Good. Works on all architectures
ValueUN = (UINTN)MyStructure;            // Good. Works on all architectures
ValueN = (INTN)MyStructure;              // Good. Works on all architectures

// 
// Casting fixed width types to pointers
//
MyStructure = (MY_STRUCTURE *)ValueU8;   // Bad.
MyStructure = (MY_STRUCTURE *)ValueU16;   // Bad.
MyStructure = (MY_STRUCTURE *)ValueU32;   // Bad. Works on 32-bit CPUs.
                                         // Works on 64-bit CPU with < 4GB
memory                                // Strips upper 32 bits on 64-bit
                                         // CPU

MyStructure = (MY_STRUCTURE *)ValueU64;   // Good. Works on all architectures
MyStructure = (MY_STRUCTURE *)Value64;   // Good. Works on all architectures
MyStructure = (MY_STRUCTURE *)ValueUN;   // Good. Works on all architectures
MyStructure = (MY_STRUCTURE *)ValueN;   // Good. Works on all architectures
```

Example 10—Examples of casting pointers

4.3.4 UEFI Data Type Sizes

Note that a few UEFI data types are different sizes on 32-bit architectures versus 64-bit architectures as follow:

- Pointers
- Enumerations
- `INTN`
- `UINTN`

The result of these differing types is that any complex types, such as unions and data structures, that are composed of these base types also have different sizes on 32-bit architectures versus 64-bit architectures. These differences must be kept in mind whenever the `sizeof()` operator is used.

If a union or data structure is required that does not change size between 32-bit and 64-bit architectures, then no changes are required.

For interoperability, only the data types defined in the Calling Conventions section of the *UEFI Specification* should be used. Some of these data types change based on the selected compiler, and should not cause a fault in the code. If a new data type is defined, then an alignment fault or other error could be generated.

4.3.5 Negative Numbers

Negative numbers are not the same on 32-bit versus 64-bit processors. Negative numbers are type `INTN`. Type `INTN` is a 4-byte container on a 32-bit processor and an 8-byte container on a 64-bit processor. For example, `-1` on a 32-bit CPU is `0xFFFFFFFF`, and `-1` on the 64-bit CPU is `0xFFFFFFFFFFFFFF`.

Caution: Be careful when assigning or comparing negative numbers. Negative numbers have different values on different architectures. For example, do not compare `-1` to `0xFFFFFFFF`, compare `-1` to `-1` and compare `0xFFFFFFFF` to `0xFFFFFFFF`. If the size of the values change, then the same compares may return different results.

The following [example](#) shows sample code that compiles without errors or warnings on both 32-bit and 64-bit architectures. However, the sample behaves very differently on 32-bit architectures versus 64-bit architectures.

```

UINT32 ValueU32;

ValueU32 = 0xFFFFFFFF;

if ((INTN)ValueU32 == -1) {
    //
    // This message is printed on 32-bit CPUs.
    // This message is not printed on 64-bit CPUs.
    //
    Print(L"Equal\n");
} else {
    //
    // This message is not printed on 32-bit CPUs.
    // This message is printed on 64-bit CPUs.
    //
    Print(L"Not Equal\n");
}

```

Example 11—Negative number example

4.3.6 Returning Pointers in a Function Parameter

The following example shows a bad example for casting pointers. The function `MyFunction()` 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 on both 32-bit and 64-bit processors. It executes on 32-bit and 64-bit processors with these unexpected side effects. It might also generate an alignment fault on IPF if `&B` is not 64-bit aligned. One possible fix for this issue is to change `B` from a `UINT32` to a `UINT64`.

```

EFI_STATUS
EFIAPI
MyFunction (
    IN  UINT32  ValueU32,
    OUT UINT64  *ValueU64
)
{
    *ValueU64 = (UINT64)ValueU32;

    return EFI_SUCCESS;
}

UINT32 A;
UINT32 B;

```

```

A = 0x11112222;
B = 0x33334444;

//
// Prints "A = 11112222 B = 33334444"
//
Print(L"A = %08x B = %08x\n",A,B);

MyFunction (A, (UINT64 *)(&B));

//
// Prints "A = 00000000 B = 11112222"
//
Print(L"A = %08x B = %08x\n",A,B);

```

Example 12—Casting OUT function parameters

4.3.7 Array Subscripts

In general, array subscripts should be of type **INTN** or **UINTN**. Using these types avoids 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 a 32-bit processor and **0x00000000FFFFFFFF** on a 64-bit processor. These subscript values are very different.

On 32-bit architectures, this value is the same indexing element as **-1** of the array. However, on a 64-bit 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 a 32-bit processor and **0xFFFFFFFFFFFFFF** on a 64-bit processor. These values are both the same indexing element as **-1** of the array.

The following example shows two array subscripts. The first one works on 32-bit architectures but accesses a very high address on 64-bit architectures that may generate a fault or hang condition. The second array subscript is rewritten to work properly on both 32-bit architectures and 64-bit architectures.

```

UINT32 Index;
CHAR8 Array[] = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
CHAR8 *MyArray;

MyArray = &(Array[5]);
Index = 0;

//
// Works on 32-bit CPUs
// Accesses high memory on 64-bit CPUs and may generate fault or hang
//
Print(L"Character = %c\n",Array[Index - 1]);

///////////////////////////////

```

```

UINTN Index;
CHAR8 Array[] = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
CHAR8 *MyArray;

MyArray = &(Array[5]);
Index = 0;

//
// Works on 32-bit CPUs and 64-bit CPUs
//
Print(L"Character = %c\n", Array[Index - 1]);

```

Example 13—Array subscripts example

4.3.8 Piecemeal Structure Allocations

Structures should always be allocated using the `sizeof()` operator on the name of the structure. Never use the sum of the sizes of the structure's members because it does not take into account the padding that the compiler introduces to guarantee alignment. The following example shows two examples for allocating memory for a structure. The first allocation is incorrect, the second allocation is correct.

```

typedef struct {
    UINT8 Value8;
    UINT64 Value64;
} EXAMPLE_STRUCTURE;

EXAMPLE_STRUCTURE *Example;

//
// Wrong. This only allocates 9 bytes, but MyStructure is 16 bytes
//
Example = AllocatePool (sizeof (UINT8) + sizeof (UINT64));

//
// Correct. This allocates 16 bytes for MyStructure.
//
Example = AllocatePool (sizeof (EXAMPLE_STRUCTURE));

```

Example 14—Incorrect and correct piecemeal structure allocation

4.4 Optimization Techniques

There are several techniques you can use to optimize a 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—sometimes they are at odds with each other. For example, a UEFI driver may grow in size to meet a specific

performance goal. The driver writer is required to make the appropriate compromises in the selection of these driver optimization techniques.

4.4.1 Size Reduction

Table 15, below, lists techniques that can be used to reduce the size of UEFI drivers. Significant size reductions can be realized by using combinations of all of these techniques. The compiler and linker switches referenced below are specific to the Microsoft™ compilers. Different compilers and linkers may use different switches for equivalent operations.

Table 15—Space optimizations

Technique	Description
MdePkg and MdeModulePkg library classes	Maximizes the use of library classes defined in the MdePkg and MdeModulePkg. In some cases, multiple implementations of the same library class may be provided. Some implementations may be smaller and others may be faster. If a UEFI Driver implementation maximizes its use of library functions from EDK II packages, then the UEFI Driver can be built with library instance mappings defined in the DSC file that minimize the size of a UEFI Driver.
Compiler flags that optimize for size.	<p>Some compiler provide flags (such as /Os and /O1) optimize code for space. This is an easy way to reduce the size of a UEFI driver.</p> <p>Note: When optimization is turned on, each line of source code may not map to the same line when a debugger is active. This can make it more difficult to debug.</p> <p>TIP: Be careful 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 or DMA bus masters.</p> <p>TIP: When optimized for speed, the UEFI driver is small, and may execute faster. If there are any speed paths in a UEFI driver that cause problems if the UEFI driver executes faster, then these switches may expose those speed paths. These same speed paths also show up as faster processors are used, so it is good to find these speed paths early.</p>
Linker flags that remove unused functions and variables	Some linkers provide flags (such as /OPT:REF) that remove 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 a UEFI driver executable. The EDK II enabled these types of flags by default.
EFI Compression	If a UEFI Driver is stored in a PCI option ROM, the UEFI compression algorithm can be used to further reduce the size of a UEFI driver. The build utility EfiRom has built-in support for compressing UEFI images. The PCI bus driver has built-in support for decompressing UEFI drivers stored in PCI option ROMs. The average compression ratio on IA32 is 2.3, and the average compression ratio on the IPF platform is 2.8. The EfiRom utility is described in Chapter 18 of this guide.

EFI Byte Code Images	If a UEFI driver is going to be stored in a PCI option ROM, and the PCI option ROM must support both IA32 and IPF platforms, or just IPF platforms, EFI Byte Code (EBC) executables should be considered. EBC executables are portable between IA32 and IPF platforms. This portability means that only a single UEFI driver image is required to support both IA32 and IPF platforms. Also, the EBC executables are significantly smaller than images for the IPF platform, so there are advantages to using this format for UEFI drivers that are targeted only at IPF platforms. In addition, using EFI Compression (see above) can reduce the EBC executables even further.
----------------------	---

4.4.2 Performance Optimizations

The following table lists the techniques to use to improve the performance of UEFI drivers. By using combinations of all of these techniques, significant performance enhancements can be realized.

Table 16—Speed optimizations

Technique	Description
Compiler flags that optimize for performance.	<p>Some compiler provide flags (such as /Ot, /O2, and /Ox) optimize code for performance. This technique is an easy way to reduce the execution time of a UEFI driver. For the most part, the EDK II balances size and speed optimizations. The flags can be customized to specify a preference for speed or size.</p> <p>Note: When optimization is turned on, each line of source code may not map to the same line when a debugger is active. This can make it more difficult to debug.</p> <p>TIP: Be careful 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.</p> <p>TIP: Because the UEFI driver is small, it may execute faster. If there are any speed paths in a UEFI driver that cause problems if the UEFI driver executes faster, then these switches may expose those speed paths. These same speed paths also show up as faster processors are used, so it is good to find these speed paths early.</p>
UEFI Services	Whenever possible, use UEFI boot services, UEFI runtime services, and the protocol services provided by other UEFI drivers. The UEFI boot services and UEFI runtime services are likely implemented as native calls that have been optimized for the platform, so there is 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 result in more efficient execution.

PCI I/O Protocol	<p>If a 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 <code>Io()</code>, <code>Mem()</code>, and <code>Pci()</code> services. See Chapter 18 for details on optimization techniques that are specific to PCI.</p>
------------------	--

4.4.3 CopyMem() and SetMem() Operations

The following example shows how to use the EDK II library `BaseMemoryLib` functions `CopyMem()` and `SetMem()` to improve the performance of a UEFI driver. These techniques apply to arrays, structures, or allocated buffers.

Note: *By default, the EDK II enables high levels of optimization, so this example may not build for all compilers because the loops are optimized into intrinsics that can cause the link to fail. So not only does use of `CopyMem()` and `SetMem()` improve performance, it also improves UEFI Driver portability.*

```
#include <Uefi.h>

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];
}
MyStructureA = MyStructureB;

// 
// Using the optimized CopyMem() Boot Services is fast
//
CopyMem(
    (VOID *)A,
    (VOID *)B,
    100
);
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.  
//  
SetMem((VOID *)A, 100, 0);  
SetMem((VOID *)&MyStructureA, sizeof (MY_STRUCTURE), 0);
```

Example 15—CopyMem() and SetMem() Speed Optimizations

5

UEFI Services

This chapter focuses on the UEFI services that apply to the implementation of UEFI drivers. This includes descriptions of those services, along with code examples, that demonstrate how a UEFI driver typically uses those services. The EDK II provides a number of library functions that simplify the use of UEFI services as well as UEFI driver improvements in maintainability, portability, readability, robustness, and size. Additional descriptions and code examples using EDK II library functions also appear where applicable.

The UEFI Boot Services and UEFI Runtime Services available to UEFI Drivers fall into three general areas:

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

The full function prototypes and descriptions for each service, and their arguments, are available in the Boot Services and Runtime Services chapters of the *UEFI Specification*. The full function prototypes and descriptions of the EDK II library functions, and their arguments, are available in the *EDK II MdePkg Package Document* and the *EDK II MdeModulePkg Package Document*.

The following table alphabetically lists all UEFI Boot and Runtime Services.

Table 17—Alphabetical listing of UEFI services

Service	Type	Service Type
AllocatePool()	Boot	Memory Allocation
AllocatePages()	Boot	Memory Allocation
CalculateCrc32()	Boot	Miscellaneous
CheckEvent()	Boot	Event
CloseEvent()	Boot	Event
CloseProtocol()	Boot	Protocol Handler
ConnectController()	Boot	Protocol Handler
ConvertPointer()	Runtime	Miscellaneous
CopyMem()	Boot	Miscellaneous
CreateEvent()	Boot	Event
CreateEventEx()	Boot	Event
DisconnectController()	Boot	Protocol Handler

Exit()	Boot	Special
ExitBootServices()	Boot	Special
FreePages()	Boot	Memory Allocation
FreePool()	Boot	Memory Allocation
GetMemoryMap()	Boot	Memory Allocation
GetNextMonotonicCount()	Boot	Special
GetNextHighMonotonicCount()	Runtime	Special
GetNextVariableName()	Runtime	Variable
GetTime()	Runtime	Time-related
GetVariable()	Runtime	Variable
GetWakeupTime()	Runtime	Time-related
HandleProtocol()	Boot	Protocol Handler
InstallConfigurationTable()	Boot	Miscellaneous
InstallMultipleProtocolInterfaces()	Boot	Protocol Handler
InstallProtocolInterface()	Boot	Protocol Handler
LoadImage()	Boot	Image
LocateDevicePath()	Boot	Protocol Handler
LocateHandle()	Boot	Protocol Handler
LocateHandleBuffer()	Boot	Protocol Handler
LocateProtocol()	Boot	Protocol Handler
OpenProtocol()	Boot	Protocol Handler
OpenProtocolInformation()	Boot	Protocol Handler
ProtocolsPerHandle()	Boot	Protocol Handler
QueryCapsuleCapabilities()	Runtime	Special
QueryVariableInfo()	Runtime	Variable
RaiseTPL()	Boot	Task Priority
RegisterProtocolNotify()	Boot	Protocol Handler
ReinstallProtocolInterface()	Boot	Protocol Handler
ResetSystem()	Runtime	Special
RestoreTPL()	Boot	Task Priority
SetMem()	Boot	Miscellaneous
SetTime()	Runtime	Time-related
SetTimer()	Boot	Time-related
SetVariable()	Runtime	Variable
SetVirtualAddressMap()	Runtime	Special
SetWakeupTime()	Runtime	Time-related

SetWatchDogTimer()	Boot	Time-related
StartImage()	Boot	Image
SignalEvent()	Boot	Event
Stall()	Boot	Time-related
UninstallMultipleProtocolInterfaces()	Boot	Protocol Handler
UninstallProtocolInterface()	Boot	Protocol Handler
UnloadImage()	Boot	Image
UpdateCapsule()	Runtime	Special
WaitForEvent()	Boot	Event

5.1 Services that UEFI drivers commonly use

The following table lists UEFI services commonly used by UEFI drivers. Following that, discussions briefly describe each service, why they are commonly used, or the particular circumstance in which they are useful. Code examples show how the services are typically used by UEFI drivers and are grouped by Service Type.

Table 18—UEFI services that are commonly used by UEFI drivers

Service	Type	Service Type	Description
AllocatePool()	Boot	Memory Allocation	Allocates a memory buffer of a particular type.
FreePool()	Boot	Memory Allocation	Frees a previously allocated memory buffer.
AllocatePages()	Boot	Memory Allocation	Allocates one memory buffer of a particular type with a 4KB aligned start address and a 4KB aligned length.
FreePages()	Boot	Memory Allocation	Frees a memory buffer previously allocated with AllocatePages().
CopyMem()	Boot	Miscellaneous	Copies a buffer from one location to another.
SetMem()	Boot	Miscellaneous	Initializes the contents of a buffer with a specified value.
InstallMultipleProtocolInterfaces()	Boot	Protocol Handler	Installs one or more protocol interfaces onto a handle. Replaces the InstallProtocolInterface() service.
UninstallMultipleProtocolInterfaces()	Boot	Protocol Handler	Uninstalls one or more protocol interfaces from a handle. Replaces the UninstallProtocolInterface() service.

LocateHandleBuffer()	Boot	Protocol Handler	Retrieves a list of handles from the handle database meeting the search criteria. The return buffer is automatically allocated.
LocateProtocol()	Boot	Protocol Handler	Finds the first handle in the handle database supporting the requested protocol.
OpenProtocol()	Boot	Protocol Handler	Adds elements to the list of agents consuming a protocol interface.
OpenProtocolInformation()	Boot	Protocol Handler	Retrieves the list of agents currently consuming a protocol interface.
CloseProtocol()	Boot	Protocol Handler	Removes elements from the list of agents consuming a protocol interface.
RaiseTPL()	Boot	Task Priority	Raises the task priority level.
RestoreTPL()	Boot	Task Priority	Restores/lowers the task priority level.
CreateEvent()	Boot	Event	Creates a general-purpose event structure.
CreateEventEx()	Boot	Event	Creates an event structure as part of an event group. <i>This service is new.</i>
CloseEvent()	Boot	Event	Closes and frees an event structure.
SignalEvent()	Boot	Event	Signals an event.
CheckEvent()	Boot	Event	Checks whether an event is in the signaled state.
SetTimer()	Boot	Time-related	Sets an event to be signaled at a particular time.
Stall()	Boot	Time-related	Waits for a specified number of microseconds. <i>This is the time-related service with the highest accuracy.</i>

5.1.1 Memory Allocation Services

The **AllocatePool()** and **FreePool()** boot 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.

The **AllocatePages()** and **FreePages()** boot 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.

5.1.1.1 Critical considerations for allocating memory

UEFI drivers should not make assumptions about the organization of system memory. Because of this, allocating from specific addresses or below specific addresses is strongly discouraged. The `AllocatePool()` service *does not* allow the caller to specify a preferred address, so this service is safe to use and does not impact the compatibility of a UEFI Driver on different platforms.

The `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 increases the compatibility of UEFI Drivers on different platforms. The allocation types of `AllocateMaxAddress` and `AllocateAddress` may reduce platform compatibility, so their use is discouraged.

Caution: *Although the Allocate services allow for specific memory allocation, do not allocate specific addresses in a UEFI driver. Allocating buffers at a specific address could result in errors, including a catastrophic failure on some platforms. Memory allocation in UEFI drivers should be done dynamically.*

TIP: Always check function return codes to verify if a memory allocation request succeeded or not before accessing the allocated buffer.

Key points:

- To prevent memory leaks, every allocation operation must have a corresponding free operation. It is important to note that some UEFI services allocate buffers for the caller and expect the caller to free those buffers.
- Buffers above 4 GB may be allocated if there is system memory present above 4 GB. As a result, all UEFI drivers must be aware that pointers may contain address values above 4 GB, and care must be taken never to strip the upper address bits.
- Structures and values placed in allocated buffers must be naturally aligned to maximize compatibility with all CPU architectures.
- Never use an allocated buffer for DMA without mapping it through an I/O Protocol. For example, the `Map()` and `UnMap()` services in the PCI I/O Protocol.

Refer to [Chapter 4](#) for general porting considerations covering more memory allocation details for 32-bit and 64-bit architectures.

5.1.1.2 Do not directly allocate a memory buffer for DMA access

A UEFI driver must *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 or if CPU caches are coherent with DMA operations. Instead, a UEFI driver must use the services provided by the I/O protocol for the bus to allocate and free buffers required for DMA operations. 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 must be provided.

5.1.1.3 Allocating and freeing buffers

UEFI boot service drivers typically allocate and free buffers of type `EfiBootServicesData`. UEFI runtime drivers typically allocate and free buffers of type `EfiRuntimeServicesData`. OS Loaders typically allocate and free buffers of type `EfiLoaderData`.

Most drivers that follow the UEFI driver model allocate private context structures in their Driver Binding Protocol `Start()` function and free them in their Driver Binding Protocol `Stop()` function. UEFI drivers may also dynamically allocate and free buffers as different I/O operations are performed.

5.1.1.4 Code examples for `AllocatePool()` and `FreePool()`

The following code fragment shows how the UEFI Boot Service `AllocatePool()` and `FreePool()` can be used to allocate and free a buffer for a data structure from `EfiBootServicesData` memory. The EDK II library `UefiBootServicesTableLib` provides global variables for the UEFI System Table, the UEFI Boot Services Table, and the Image Handle for the currently executing driver. In this example, the global variable for the UEFI Boot Services Table, called `gBS`, is used to call the UEFI Boot Services `AllocatePool()` and `FreePool()`.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS      Status;
EXAMPLE_DEVICE *Device;

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

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

Example 16—Allocate and free pool using UEFI Boot Services Table

The [code fragment](#) below shows exactly the same functionality as Example 16, above, but uses EDK II library `MemoryAllocationLib` to simplify the implementation. The `MemoryAllocationLib` function `AllocatePool()` allocates memory of type `EfiBootServicesData`. If memory of type `EfiRuntimeServicesData` is required, then the `MemoryAllocationLib` function `AllocateRuntimePool()` should be used.

```
#include <Uefi.h>
#include <Library/MemoryAllocationLib.h>

EXAMPLE_DEVICE *Device;

//
// Allocate a buffer for a data structure
//
Device = (EXAMPLE_DEVICE *)AllocatePool (sizeof (EXAMPLE_DEVICE));
if (Device == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

//
// Free the allocated buffer
//
FreePool (Device);
```

Example 17—Allocate and free pool using MemoryAllocationLib

In many cases, when a structure is allocated, it is useful to clear the buffer to a known state with zeros. The following code fragment in Example 18 expands on Example 17, above, showing how the EDK II library **MemoryAllocationLib** can be used to allocate and clear a buffer in a single call.

```
#include <Uefi.h>
#include <Library/MemoryAllocationLib.h>

EXAMPLE_DEVICE *Device;

//
// Allocate and clear a buffer for a data structure
//
Device = (EXAMPLE_DEVICE *)AllocateZeroPool (sizeof (EXAMPLE_DEVICE));
if (Device == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

//
// Free the allocated buffer
//
FreePool (Device);
```

Example 18—Allocate and clear pool using MemoryAllocationLib

Complex structures that require many fields to be initialized after the structure is allocated may increase the size of the UEFI driver if the fields are initialized one by one. The EDK II library **MemoryAllocationLib** provides an additional allocation method that makes use of a template structure to reduce code size.

The concept is that a template structure can be declared as a global variable with all the fields pre-initialized to the required values. It takes less space to store just the data than it does to store the instructions and data to initialize all the fields one by one. This technique may be useful for UEFI Drivers that produce new protocols for each device the UEFI Driver manages. [Example 19](#), below, expands on the above Example

18 showing how the EDK II library `MemoryAllocationLib` is used to allocate and initialize a buffer from a template structure in a single call.

```
#include <Uefi.h>
#include <Library/MemoryAllocationLib.h>

EXAMPLE_DEVICE gExampleDeviceTemplate = {
    EXAMPLE_PRIVATE_DATA_SIGNATURE,
    //
    // Other device specific fields
    //
};

EXAMPLE_DEVICE *Device;

//
// Allocate and initialize a buffer for a data structure
//
Device = (EXAMPLE_DEVICE *)AllocateCopyPool (
    sizeof (EXAMPLE_DEVICE),
    &gExampleDeviceTemplate
);
if (Device == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

//
// Free the allocated buffer
//
FreePool (Device);
```

Example 19—Allocate and initialize pool using `MemoryAllocationLib`

5.1.1.5 Code examples for `AllocatePages()` and `FreePages()`

The following code fragment shows how the UEFI Boot Services `AllocatePages()` and `FreePages()` are used to allocate and free a 16KB buffer for a data structure from `EfiBootServicesData` memory. The EDK II library `UefiBootServicesTableLib` provides global variables for the UEFI System Table, the UEFI Boot Services Table, and the Image Handle for the currently executing driver. In this example, the global variable for the UEFI Boot Services Table, called `gBS`, is used to call the UEFI Boot Services `AllocatePages()` and `FreePages()`.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS Status;
EFI_PHYSICAL_ADDRESS PhysicalBuffer;
UINTN Pages;
VOID *Buffer;

//
// Allocate the number of pages to hold Size bytes and
// return in PhysicalBuffer
//
Pages = EFI_SIZE_TO_PAGES (SIZE_16KB);
```

```

Status = gBS->AllocatePages(
    AllocateAnyPages,
    EfiBootServicesData,
    Pages,
    &PhysicalBuffer
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Convert the physical address to a pointer.
// This method works for all support CPU architectures.
//
Buffer = (VOID *)(UINTN)PhysicalBuffer;

//
// Free the allocated buffer
//
Status = gBS->FreePages (PhysicalBuffer, Pages);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 20—Allocate and free pages using UEFI Boot Services Table

The code fragment in Example 21, below, shows the same functionality as Example 20, above, but uses the EDK II library `MemoryAllocationLib` to simplify the implementation. The `MemoryAllocationLib` function `AllocatePages()` allocates memory of type `EfiBootServicesData`. If memory of type `EfiRuntimeServicesData` is required, the `MemoryAllocationLib` function `AllocateRuntimePages()` should be used.

```

#include <Uefi.h>
#include <Library/MemoryAllocationLib.h>

EXAMPLE_DEVICE *Device;
UINTN          Pages;

//
// Allocate a buffer for a data structure
//
Pages = EFI_SIZE_TO_PAGES (sizeof (EXAMPLE_DEVICE));
Device = (EXAMPLE_DEVICE *)AllocatePages (Pages);
if (Device == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

//
// Free the allocated buffer
//
FreePages (Device, Pages);

```

Example 21—Allocate and free pages using MemoryAllocationLib

In some rare circumstances, a UEFI Driver may be required to allocate a buffer with a specific alignment. `AllocatePool()` provides 8-byte alignment. `AllocatePages()` provides 4KB alignment. If an alignment above 4KB is required, the preferred technique is to

allocate a large buffer through `AllocatePages()`, find the portion of the allocated buffer that meets the required alignment, and free the unused portions. EDK II library `MemoryAllocationLib` provides the function called `AllocateAlignedPages()` that implements this technique. The code fragment in the example below allocates a 16KB buffer aligned on a 64KB boundary.

```
#include <Uefi.h>
#include <Library/MemoryAllocationLib.h>

VOID    *Buffer;
UINTN   Pages;

//
// Allocate a buffer for a data structure
//
Pages = EFI_SIZE_TO_PAGES (SIZE_16KB);
Buffer = (EXAMPLE_DEVICE *)AllocateAlignedPages (Pages, SIZE_64KB);
if (Buffer == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

//
// Free the allocated buffer
//
FreePages (Buffer, Pages);
```

Example 22—Allocate and free aligned pages using `MemoryAllocationLib`

5.1.2 Miscellaneous Services

The `SetMem()` and `CopyMem()` UEFI Boot Services are used by UEFI drivers to initialize the contents of a buffer or copy a buffer from one location to another. The `SetMem()` service is most commonly used to fill the contents of a buffer with zeros after it is allocated. The `CopyMem()` service handles buffers of any alignment and also handles the rare case when the source and destination buffer overlap. With overlapping buffers, the requirement is that the destination buffer on exit from this service must match the contents of the source buffer on entry to this service.

The code fragments in this section also show examples that use the EDK II library class `BaseMemoryLib` as an alternative to using the UEFI Boot Services directly. The advantage of using this library class is that the source code can be implemented just once. The EDK II DSC file used to build a UEFI Driver can specify mappings to different implementations of the `BaseMemoryLib` library class that meet the requirements of a specific target. For example, the `MdePkg/Library/UefiMemoryLib` library instance uses the recommended UEFI Boot Services `SetMem()` and `CopyMem()` are for best performance when building a UEFI Driver for EBC. For best performance on IA32 or X64, use the SSE2 optimized `MdePkg/Library/BaseMemoryLibSse2` library instance.

5.1.2.1 Code examples for `SetMem()`

Use the `SetMem()` UEFI Boot Service 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 also be used to fill a buffer with other values. The following code fragment in the [example below](#) shows the same example from [Example 16](#), but uses `SetMem()` UEFI

Boot Service to zero the contents of the allocated buffer. The EDK II library **UefiBootServicesTableLib** provides global variables for the UEFI System Table, the UEFI Boot Services Table, and the Image Handle for the currently executing driver. Here, the global variable for the UEFI Boot Services Table called **gBS** is used to call the UEFI Boot Services **AllocatePool()** and **SetMem()**.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS      Status;
EXAMPLE_DEVICE *Device;

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

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

Example 23—Allocate and clear a buffer using UEFI Boot Services

The following code fragment shows the same example from [Example 17](#), but uses the **SetMem()** function from the EDK II library class **BaseMemoryLib** to zero the contents of the allocated buffer.

```
#include <Uefi.h>
#include <Library/MemoryAllocationLib.h>
#include <Library/BaseMemoryLib.h>

EXAMPLE_DEVICE *Device;

//
// Allocate a buffer for a data structure
//
Device = (EXAMPLE_DEVICE *)AllocatePool (sizeof (EXAMPLE_DEVICE));
if (Device == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

//
// Zero the contents of the allocated buffer
//
SetMem (Device, sizeof (EXAMPLE_DEVICE), 0);
```

Example 24—Allocate and clear a buffer using BaseMemoryLib

The code fragment in Example 25, below, shows the same example from [Example 17](#), above, but uses the `ZeroMem()` function from the EDK II library class `BaseMemoryLib` to zero the contents of the allocated buffer.

```
#include <Uefi.h>
#include <Library/MemoryAllocationLib.h>
#include <Library/BaseMemoryLib.h>

EXAMPLE_DEVICE *Device;

//
// Allocate a buffer for a data structure
//
Device = (EXAMPLE_DEVICE *)AllocatePool (sizeof (EXAMPLE_DEVICE));
if (Device == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

//
// Zero the contents of the allocated buffer
//
ZeroMem (Device, sizeof (EXAMPLE_DEVICE));
```

Example 25—Allocate and clear a buffer using `BaseMemoryLib`

5.1.2.2 Code examples for `CopyMem()`

The following code fragment shows an example of how the `CopyMem()` UEFI Boot Service is typically used to copy an existing buffer into a newly allocated buffer. The `AllocatePool()` function from the EDK II library `MemoryAllocationLib` is used to allocate a new buffer. The EDK II library `UefiBootServicesTableLib` provides global variables for the UEFI System Table, the UEFI Boot Services Table, and the Image Handle for the currently executing driver. In this example, the global variable for the UEFI Boot Services Table called `gBS` is used to call the UEFI Boot Service `CopyMem()`.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>

EXAMPLE_DEVICE *SourceDevice;
EXAMPLE_DEVICE *Device;

//
// Allocate a buffer for a data structure
//
Device = (EXAMPLE_DEVICE *)AllocatePool (sizeof (EXAMPLE_DEVICE));
if (Device == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

//
// Copy contents of SourceDevice to the allocated Device
//
gBS->CopyMem (Device, SourceDevice, sizeof (EXAMPLE_DEVICE));
```

Example 26—Allocate and copy buffer

The code fragment in Example 27, below, shows the same example from Example 26, above, but uses the `CopyMem()` function from the EDK II library class `BaseMemoryLib` to copy the contents of an existing buffer to a newly allocated buffer.

```
#include <Uefi.h>
#include <Library/MemoryAllocationLib.h>
#include <Library/BaseMemoryLib.h>

EXAMPLE_DEVICE *SourceDevice;
EXAMPLE_DEVICE *Device;

//
// Allocate a buffer for a data structure
//
Device = (EXAMPLE_DEVICE *)AllocatePool (sizeof (EXAMPLE_DEVICE));
if (Device == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

//
// Copy contents of SourceDevice to the allocated Device
//
CopyMem (Device, SourceDevice, sizeof (EXAMPLE_DEVICE));
```

Example 27—Allocate and clear a buffer using BaseMemoryLib

5.1.3 Handle Database and Protocol Services

There are several UEFI Boot Services used to add, retrieve, and remove contents from the Handle Database. Concepts of the Handle Database and Protocols are introduced in [Section 3.4](#). This section provides code examples for the UEFI Boot Services commonly used by UEFI Drivers to manage the Handle Database and include the following:

- `InstallMultipleProtocolInterfaces()`
- `UninstallMultipleProtocolInterfaces()`
- `LocateHandleBuffer()`
- `LocateProtocol()`
- `OpenProtocol()`
- `OpenProtocolInformation()`
- `CloseProtocol()`

5.1.3.1 `InstallMultipleProtocolInterfaces()` and `UninstallMultipleProtocolInterfaces()`

These services are used to do the following:

- Create new handles in the Handle Database.

- Remove a handle from the Handle Database.
- Add protocols to an existing handle in the Handle Database.
- Remove protocols from an existing handle in the Handle Database.

Extra services to create a new handle in the Handle Database and remove an existing handle from the Handle Database are not required. Instead, the first protocol installed onto a handle automatically creates a new handle and adds that handle to the Handle Database. The last protocol removed from an existing handle automatically removes the handle from the Handle Database and destroys the handle. This means it is not possible for a handle to be present in the Handle Database with zero protocols installed.

Another important concept is that a single handle in the Handle Database is not allowed to have more than one instance of the same Protocol installed onto that handle. If a UEFI Driver is required to produce more than one instance of the same protocol, then the Protocol instances must be installed on different handles in the Handle Database.

UEFI Drivers tend to manage more than one protocol at a time. Because of this, it is recommended that `InstallMultipleProtocolInterfaces()` and `UninstallMultipleProtocolInterfaces()` be used instead of the `InstallProtocolInterface()` and `UninstallProtocolInterface()`. This results in source code that is easier to maintain and also tends to produce smaller executables. In addition, `InstallMultipleProtocolInterfaces()` provides more extensive error checking than `InstallProtocolInterface()`, which allows developers to catch coding errors sooner, and results in higher quality UEFI Driver implementations. The main difference is that `InstallMultipleProtocolInterfaces()` guarantees that a duplicate Device Path Protocol is never be added to the Handle Database. [Section 3.9](#) introduces the concept of Device Paths and the requirement for them to be unique.

The `InstallMultipleProtocolInterfaces()` and `UninstallMultipleProtocolInterfaces()` services support adding and removing more than one protocol at a time through the use of a variable argument list. Protocols are represented by a pair of pointers to a protocol GUID and a protocol interface. These services parse pairs of arguments until a `NULL` pointer for the protocol GUID parameter is encountered.

Note: *If any errors are generated when the protocols are being added to a handle, any protocols added before the error is returned, are automatically removed by `InstallMultipleProtocolInterfaces()`. This means the state of the handle in the handle database is identical to the state prior to the call.*

Note: *If any errors are generated when the protocols are being removed from a handle, any protocols removed before the error is returned, are automatically added back by `UninstallMultipleProtocolInterfaces()`. This means the state of the handle in the handle database is identical to the state prior to the call.*

TIP: If unexpected errors are returned by these services, try converting a single call for multiple protocols to a series of calls that process one protocol at a time. This allows the specific protocol causing the error condition to be identified. It should be rare for these services to return an error condition. If an error condition is present it is likely due to a duplicate GUID, a duplicate device path, or an invalid handle.

Note: When an attempt is made to remove a protocol interface from a handle in the handle database, the UEFI core firmware checks to see if any other UEFI drivers are currently using the services of the protocol to be removed. If UEFI drivers are using that protocol interface, the UEFI core firmware attempts to stop those UEFI drivers with a call to `DisconnectController()`. This is a quick, legal, and safe way to shut down any protocols associated with this driver's stack.

Caution: A serious issue can occur when a user removes and then reattaches a device on a bus that supports hot-plugging. Driver writers must consider this when writing drivers for hot-plug devices.

The issue occurs when other controllers are also using one, or more, of a driver's protocols. In these cases, the `UninstallMultipleProtocolInterfaces` service fails.

If the call to `DisconnectController()` fails, the UEFI core firmware then calls `ConnectController()` to put the handle database back into the same state that it was in prior to the original call to `UninstallMultipleProtocolInterfaces()`. This call to `ConnectController()` has the potential to cause issues upon re-entry in UEFI drivers that must be handled in the UEFI driver. These issues could include lost or missed connected pointer linkages resulting in lost data, confused operation, crashes and other errors. See [Chapter 31](#) in this guide for recommendations on how to test UEFI drivers.

5.1.3.1.1 Protocols that may be added at the driver entry point

The following protocols may also be added in the driver entry point with the `InstallMultipleProtocolInterfaces()` service. Please see [Chapter 7](#) for more details on how to install these protocols in a driver entry point along with the recommendations on when each of these protocols should be installed in a driver entry point. Later chapters of this guide cover the implementation of these protocols in more detail.

- Driver Binding Protocol
- Component Name Protocol
- Component Name 2 Protocol
- Driver Configuration Protocol
- Driver Configuration 2 Protocol
- Driver Diagnostics Protocol
- Driver Diagnostics 2 Protocol
- HII Config Access Protocol
- Driver Health Protocol
- Driver Family Override Protocol
- Driver Supported EFI Version Protocol

5.1.3.1.2 Removing protocols when a driver is unloaded

If a UEFI driver is unloadable, then the protocols that were added in the driver entry point must be removed in the driver's `Unload()` function using `UninstallMultipleProtocolInterfaces()`.

TIP: Although the `Unload()` function is optional, uninstalling the protocols in the `Unload()` function of a driver is not optional. The install and uninstall sections must mirror each other for the protocols used by the driver.

TIP: The `load` and `unload` UEFI Shell commands may be used to test driver load and unload services for handles and protocols.

5.1.3.1.3 Code example

The following code fragment shows how `InstallMultipleProtocolInterfaces()` can be used from the entry point of a UEFI Driver to install driver related protocols. This example installs the Driver Binding Protocol, required for UEFI Drivers that follow the UEFI Driver Model, along with the Component Name 2 Protocol which is optional for UEFI Drivers that follow the UEFI Driver Model. Both protocols are installed onto the image handle passed into the entry point of the UEFI Driver, and the call to `InstallMultipleProtocolInterfaces()` uses GUID/Pointer pairs terminated by a `NULL` GUID value. Additional optional protocols could be added to this one call to `InstallMultipleProtocolInterfaces()` depending on a specific UEFI Driver requirements and capabilities.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>

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

EFI_COMPONENT_NAME2_PROTOCOL gMyComponentName2 = {
    MyGetDriverName,
    MyGetControllerName,
    "en"
};

EFI_STATUS Status;
EFI_HANDLE ImageHandle;

//
// Install the Driver Binding Protocol and the Component Name 2 Protocol
// onto the image handle that is passed into the driver entry point
//
Status = gBS->InstallMultipleProtocolInterfaces (
    &ImageHandle,
    &gEfiDriverBindingProtocolGuid,  &gMyDriverBinding,
    &gEfiComponentName2ProtocolGuid, &gMyComponentName2,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 28—Install protocols in UEFI Driver entry point.

The code fragment in [Example 29](#) performs the same work as the example above, but uses the EDK II **UefiLib** to install the UEFI Driver Model related protocols. In this specific case, the Driver Binding Protocol, Component Name Protocol, and Component Name 2 Protocols are all installed using the **UefiLib** function **EfiLibInstallDriverBindingComponentName2()**. The Component Name Protocol and Component Name 2 Protocol implementations use the same functions for their protocol implementations, thereby reducing the size overhead for supporting both name protocols.

The EDK II **UefiLib** provides 4 functions that may be used to initialize a UEFI Driver that follows the UEFI Driver Model. The Component Name Protocols are declared with **GLOBAL_REMOVE_IF_UNREFERENCED** that guarantees the protocol structures are removed from the final binary UEFI Driver image if the EDK II build is configured to not produce the Component Name Protocols. It does not make sense to use that declaration style for the Driver Binding Protocol since that protocol must always be produced by a UEFI Driver that follows the UEFI Driver Model.

The EDK II library **UefiLib** uses several Platform Configuration Database (PCD) feature flags to enable and disable the Component Name Protocols at build time. [Chapter 30](#) covers how to build UEFI Drivers in the EDK II and also covers configuration of UEFI Drivers through PCD settings.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>

#define MY_VERSION 0x10

EFI_DRIVER_BINDING_PROTOCOL gMyDriverBinding = {
    MySupported,
    MyStart,
    MyStop,
    MY_VERSION,
    NULL,
    NULL
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gMyComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME) MyGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) MyGetControllerName,
    "eng"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gMyComponentName2 = {
    MyGetDriverName,
    MyGetControllerName,
    "en"
};

EFI_STATUS Status;
EFI_HANDLE ImageHandle;

//
```

```

// Install driver model protocol(s).
//
Status = EfiLibInstallDriverBindingComponentName2 (
    ImageHandle,
    SystemTable,
    &gMyDriverBinding,
    ImageHandle,
    &gMyComponentName
    &gMyComponentName2
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 29—Install protocols in UEFI Driver entry point using UefiLib.

The code fragment below shows how the protocols installed in the previous example would be uninstalled in a UEFI Driver's `Unload()` function. A UEFI Driver is not required to implement the `Unload()` capability, but if the `Unload()` capability is implemented, it must undo the work performed in the entry point of the UEFI Driver just like `InstallMultipleProtocolInterfaces()`. `UninstallMultipleProtocolInterfaces()` allows multiple protocols to be specified in a single call using a set of GUID/Pointer arguments terminated by a `NULL` GUID value.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>

EFI_STATUS Status;
EFI_HANDLE ImageHandle;

//
// 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,
    &gEfiComponentName2ProtocolGuid, &gMyComponentName2,
    &gEfiComponentNameProtocolGuid,  &gMyComponentName,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 30—Uninstall protocols in UEFI Driver `Unload()` function.

UEFI device drivers add protocols for I/O services to existing handles in the handle database in their `Start()` function and 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 typically adds the `EFI_DEVICE_PATH_PROTOCOL` and an I/O abstraction for the bus type managed by that bus driver. For example, the PCI bus driver creates 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 if the bus type supports a standard container for storing UEFI Drivers.

The following code fragment shows an example of how a child handle can be added to the handle database with a Device Path Protocol and then add a Block I/O Protocol to that same child handle. These two operations could also be combined into a single call to `InstallMultipleProtocolInterfaces()`.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/DevicePath.h>
#include <Protocol/BlockIo.h>

EFI_STATUS Status;
EFI_HANDLE ChildHandle;
EFI_DEVICE_PATH_PROTOCOL *DevicePath;
EFI_BLOCK_IO_PROTOCOL *BlockIo;

// Add Device Path Protocol to a new handle
// ChildHandle = NULL;
Status = gBS->InstallMultipleProtocolInterfaces (
    &ChildHandle,
    &gEfiDevicePathProtocolGuid, DevicePath,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}

// Add the Block I/O Protocol to the handle created in the previous call
// Status = gBS->InstallMultipleProtocolInterfaces (
//     &ChildHandle,
//     &gEfiBlockIoProtocolGuid, BlockIo,
//     NULL
// );
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 31—Add child handle to handle database

The following [code fragment below](#) shows an example of how the child handle created in the previous example can be destroyed by uninstalling all the installed protocols in a single call to `UninstallMultipleProtocolInterfaces()`.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/DevicePath.h>
#include <Protocol/BlockIo.h>

EFI_STATUS Status;
EFI_HANDLE ChildHandle;
EFI_DEVICE_PATH_PROTOCOL *DevicePath;
EFI_BLOCK_IO_PROTOCOL *BlockIo;

//
// Remove Device Path Protocol and Block I/O Protocol from the child
// handle created above. Because this call removes all the
// protocols from the handle, the handle is removed from the
// handle database.
//
Status = gBS->UninstallMultipleProtocolInterfaces (
    ChildHandle,
    &gEfiDevicePathProtocolGuid, DevicePath,
    &gEfiBlockIoProtocolGuid, BlockIo,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 32—Remove child handle from handle database.

A more rare use case of `InstallMultipleProtocolInterfaces()` is installing a protocol with a `NULL` protocol interface pointer. The GUID value in this case is called a tag GUID because there are no data fields or services associated with the GUID.

The code fragment below shows an example of adding a tag GUID to the handle of a controller that a UEFI Driver is managing. In this example, the tag GUID used is the GUID name of the UEFI Driver itself called `gEfiCallerIdGuid`.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS Status;
EFI_HANDLE ControllerHandle;

//
// Add tag GUID called gEfiCallerIdGuid to ControllerHandle
//
Status = gBS->InstallMultipleProtocolInterfaces (
    &ControllerHandle,
    &gEfiCallerIdGuid, NULL,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 33—Add tag GUID to a controller handle.

The following code fragment shows how the tag GUID added in the previous example can be removed.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS Status;
EFI_HANDLE ControllerHandle;

//
// Remove tag GUID called gEfiCallerIdGuid from ControllerHandle
//
Status = gBS->UninstallMultipleProtocolInterfaces (
    ControllerHandle,
    &gEfiCallerIdGuid, NULL,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 34—Remove tag GUID from a controller handle.

5.1.3.2 LocateHandleBuffer()

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

- Retrieve **AllHandles**: Retrieve all handles in the handle database.
- Retrieve **ByProtocol**: Retrieve all handles in the handle database that support a specified protocol.
- Retrieve **ByRegisterNotify**: Retrieve the handle for which a specific protocol was just installed and configured for register notification using **RegisterProtocolNotify()**. This search option is strongly discouraged for UEFI Drivers. It was used with previous releases of the *EFI Specification* before the introduction of the *UEFI Driver Model*.

The buffer returned by **LocateHandleBuffer()** is allocated by the service **AllocatePool()**. A UEFI driver using this service is responsible for freeing the returned buffer when the UEFI driver no longer requires its contents use the service **FreePool()**.

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

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>

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 which
// is allocated using AllocatePool().
//
Status = gBS->LocateHandleBuffer (
    AllHandles,
    NULL,
    NULL,
    &HandleCount,
    &HandleBuffer
);
if (EFI_ERROR (Status)) {
    return Status;
}

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

Example 35—Retrieve all handles in handle database

The code fragment below 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 `OpenProtocol()`.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>
#include <Protocol/BlockIo.h>

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
// which is allocated using AlocatePool()
//
```

```

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 Block I/O Protocol instance
// from each handle.
//
for (Index = 0; Index < HandleCount; Index++) {
    Status = gBS->OpenProtocol (
        HandleBuffer[Index],
        &gEfiBlockIoProtocolGuid,
        (VOID **) &BlockIo,
        gImageHandle,
        NULL,
        EFI_OPEN_PROTOCOL_GET_PROTOCOL
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    //
    // BlockIo can be used here to make Block I/O Protocol
    // service requests.
    //
}

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

```

Example 36—Retrieve all Block I/O Protocols in handle database

5.1.3.3 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 *UEFI Specification* defines the following service protocols:

- **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL**
- **EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL**
- **EFI_UNICODE_COLLATION_PROTOCOL**
- **EFI_DEBUG_SUPPORT_PROTOCOL**
- **EFI_DECOMPRESS_PROTOCOL**
- **EFI_ACPI_TABLE_PROTOCOL**

- `EFI_EBC_PROTOCOL`
- `EFI_BIS_PROTOCOL`
- `EFI_KEY_MANAGEMENT_SERVICE_PROTOCOL`
- `EFI_HII_FONT_PROTOCOL`
- `EFI_HII_STRING_PROTOCOL`
- `EFI_HII_IMAGE_PROTOCOL`
- `EFI_HII_DATABASE_PROTOCOL`
- `EFI_HII_CONFIG_ROUTING_PROTOCOL`
- `EFI_FORM_BROWSER2_PROTOCOL`
- `EFI_USER_MANAGER_PROTOCOL`
- `EFI_DEFERRED_IMAGE_LOAD_PROTOCOL`
- `EFI_FIRMWARE MANAGEMENT_PROTOCOL`

This service also supports retrieving protocols that have been notified with `RegisterProtocolNotify()`, but use of `RegisterProtocolNotify()` is discouraged in UEFI Drivers, so this use case of `LocateProtocol()` is not covered. See [Section 5.3.6](#) for more details on `RegisterProtocolNotify()`.

The following code fragment shows how the `LocateProtocol()` service can be used to retrieve the first instance of a service protocol in the handle database. In this example the `EFI_DECOMPRESS_PROTOCOL` is used.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/Decompress.h>

EFI_STATUS Status;
EFI_DECOMPRESS_PROTOCOL *Decompress;

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

Example 37—Locate first Decompress Protocol in handle database

5.1.3.4

OpenProtocol() and CloseProtocol()

The `OpenProtocol()` and `CloseProtocol()` services are used by UEFI drivers to acquire and release the protocol interfaces from the handle database that the UEFI drivers require to produce their services. The `OpenProtocol()` service is one of the most complex UEFI Boot Services because it is required to support all of the various UEFI Driver types. UEFI applications and UEFI OS loaders may also use these services to lookup and use protocol interfaces in the handle database.

Caution: Proper use of `openProtocol()` and `CloseProtocol()` is required for interoperability with other UEFI components. There are UEFI Shell commands that may be used to help verify the proper use of these services including `dh`, `connect`, `disconnect`, `reconnect`, `drivers`, `devices`, `devtree`, and `openinfo`.

`OpenProtocol()` is typically used by the `Supported()` and `Start()` functions of a UEFI driver to retrieve protocol interface(s) that are installed on handles in the handle database. The code, below, shows the full function prototype for the UEFI Boot Service `OpenProtocol()`.

The `closeProtocol()` service removes an element from the list of agents that are consuming a protocol interface. UEFI drivers must close each protocol they open when the UEFI Driver no longer requires the use of that protocol. Closing protocols is typically done in the `Stop()` function.

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

/**
 * Queries a handle to determine if it supports a specified protocol. If
 * the protocol is supported by the handle, it opens the protocol on
 * behalf of the calling agent.
 *
 * @param Handle The handle for the protocol interface
 * that is being opened.
 * @param Protocol The published unique identifier of the
 * protocol.
 * @param Interface Supplies the address where a pointer to
 * the corresponding Protocol Interface is
 * returned.
 * @param AgentHandle The handle of the agent that is opening
 * the protocol interface specified by
 * Protocol and Interface.
 * @param ControllerHandle If the agent that is opening a protocol
 * is a driver that follows the UEFI Driver
 * Model, then this parameter is the
 * controller handle that requires the
 * protocol interface. If the agent does not
 * follow the UEFI Driver Model, then this
 * parameter is optional and may be NULL.
 * The open mode of the protocol interface
 * specified by Handle and Protocol.
 * @param Attributes An item was added to the open list for
 * the protocol interface, and the protocol
 * interface was returned in Interface.
 * @retval EFI_SUCCESS Handle does not support Protocol.
 * @retval EFI_UNSUPPORTED One or more parameters are invalid.
 * @retval EFI_ACCESS_DENIED Required attributes can't be supported in
 * current environment.
 * @retval EFI_ALREADY_STARTED Item on the open list already has
 * required attributes whose agent handle is
 * the same as AgentHandle.
 */
```

```

**/
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
);


```

Example 38—OpenProtocol() function prototype

The `Handle` and `Protocol` parameters specify “what” protocol interface is being opened. The `AgentHandle` and `ControllerHandle` specifies “who” is opening the protocol interface. The `Attributes` parameter specifies “why” a protocol interface is being opened. The `Interface` parameter is used to return the protocol interface if it is successfully opened, and the `EFI_STATUS` return code tells if the protocol interface was opened or not and if not, why it could not be opened. The UEFI core records these input parameter values to track how each protocol interface is being used. This tracking information can be retrieved through the `OpenProtocolInformation()` service. The `EFI_STATUS` code returned by `OpenProtocol()` is very important and must be examined by UEFI drivers that use this service. In some cases, error code such as `EFI_ALREADY_STARTED` may be the expected result and may not be an error at all for that specific UEFI Driver.

Caution: Make sure that all status codes returned by `OpenProtocol()` are properly evaluated.

`AgentHandle` and `ControllerHandle` describe “who” is opening the protocol interface. For UEFI drivers, the `AgentHandle` parameter is typically the `DriverBindingHandle` field from the `EFI_DRIVER_BINDING_PROTOCOL` produced by the UEFI Driver. UEFI Drivers that are device drivers producing additional protocols on the same handle typically use the same value for `Handle` and `ControllerHandle`. UEFI Drivers that are bus drivers producing child handles may use `OpenProtocol()` with `Handle` set to the handle for the bus controller and `ControllerHandle` set to the handle of a child controller.

The `Attributes` parameter is a bitmask that describes “why” the protocol interface is being opened. The `#define` values used to build an `Attributes` value are also shown in [Example 38](#) above. They are the `#define` statements. A summary of the attribute combinations used by UEFI drivers is listed below.

Caution: Make sure UEFI Drivers use the attributes correctly. If the attributes are used incorrectly, a driver may not function properly and may cause problems with other drivers. There are UEFI Shell commands to help verify the proper use of attributes including `dh`, `connect`, `disconnect`, `reconnect`, `drivers`, `devices`, `devtree`, and `openinfo`.

`EFI_OPEN_PROTOCOL_TEST_PROTOCOL`

Tests to see if a protocol interface is present on a handle. Typically used in the `Supported()` service of a UEFI driver if the services of the protocol being tested are not required to complete the support check.

`EFI_OPEN_PROTOCOL_GET_PROTOCOL`

Retrieves a protocol interface from a handle. Typically used in the `Supported()` and `start()` services of a UEFI driver to make use of the services of a protocol that is allowed to be used by more than one UEFI Driver.

`EFI_OPEN_PROTOCOL_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. Typically used in the `Supported()` and `Start()` services of a UEFI driver to use the services of a protocol that is not allowed to be used by more than one UEFI Driver.

`EFI_OPEN_PROTOCOL_EXCLUSIVE`

Typically used by UEFI Applications to gain exclusive access to a protocol interface. If any drivers have the same protocol interface opened with an attribute of `EFI_OPEN_PROTOCOL_BY_DRIVER`, then an attempt is made to remove them by calling `Stop()` function in that UEFI Driver. If a UEFI Driver opens a protocol interface with this attribute, no other drivers are allowed to open the same protocol interface with the `EFI_OPEN_PROTOCOL_BY_DRIVER` attribute. This attribute is used very rarely.

TIP: For good coding practices, UEFI Drivers that require the use of the `EFI_OPEN_PROTOCOL_EXCLUSIVE` attribute should combine it with the `EFI_OPEN_PROTOCOL_BY_DRIVER` attribute.

`EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_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 is not released until the driver that opened this attribute chooses to close it. This attribute is used very rarely.

`EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER`

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.

`EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL`

Do not use from a UEFI Driver. Only provided for backwards compatibility with older versions of the *EFI Specification*. Use `EFI_OPEN_PROTOCOL_GET_PROTOCOL` instead.

5.1.3.4.1 Using `EFI_OPEN_PROTOCOL_TEST_PROTOCOL`

The code fragment below tests for the presence of the PCI I/O Protocol using the `EFI_OPEN_PROTOCOL_TEST_PROTOCOL` attribute. When this attribute is used, the protocol does not have to be closed because a protocol interface is not returned when this open mode is used.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_DRIVER_BINDING_PROTOCOL *This;
EFI_HANDLE ControllerHandle;

//
// Test to see if ControllerHandle supports the PCI I/O Protocol
//

```

```

Status = gBS->OpenProtocol (
    ControllerHandle,           // Handle
    &gEfiPciIoProtocolGuid,     // Protocol
    NULL,                      // Interface
    This->DriverBindingHandle, // AgentHandle
    ControllerHandle,          // ControllerHandle
    EFI_OPEN_PROTOCOL_TEST_PROTOCOL // Attributes
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 39—OpenProtocol() TEST_PROTOCOL

5.1.3.4.2 Using `EFI_OPEN_PROTOCOL_GET_PROTOCOL`

The following code fragment shows the same example as above but retrieves the PCI I/O Protocol using the `EFI_OPEN_PROTOCOL_GET_PROTOCOL` attribute. With this attribute, the protocol does not have to be closed.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_DRIVER_BINDING_PROTOCOL *This;
EFI_HANDLE ControllerHandle;

// 
// Retrieve PCI I/O Protocol interface on ControllerHandle
//

Status = gBS->OpenProtocol (
    ControllerHandle,           // Handle
    &gEfiPciIoProtocolGuid,     // Protocol
    NULL,                      // Interface
    This->DriverBindingHandle, // AgentHandle
    ControllerHandle,          // ControllerHandle
    EFI_OPEN_PROTOCOL_GET_PROTOCOL // Attributes
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 40—OpenProtocol() GET_PROTOCOL

Caution: It can be dangerous to use this open mode (in which a protocol does not have to be closed) because a protocol may be removed at any time without notifying the UEFI Driver that used this mode. This means that a driver using `EFI_OPEN_PROTOCOL_GET_PROTOCOL` may attempt to use a stale protocol interface pointer that is no longer valid.

TIP: Use `EFI_OPEN_PROTOCOL_BY_DRIVER` first, to prevent the protocol from being removed while a driver is using the protocol.

The `EFI_OPEN_PROTOCOL_GET_PROTOCOL` attribute can then be used to retrieve the needed protocol interface.

A UEFI driver should be designed to use `EFI_OPEN_PROTOCOL_BY_DRIVER` as its first choice. However, there are cases where a different UEFI driver has already opened the protocol that is required by `EFI_OPEN_PROTOCOL_BY_DRIVER`. In these cases, use `EFI_OPEN_PROTOCOL_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 with `EFI_OPEN_PROTOCOL_BY_DRIVER`.

If a layer needs to skip a layer to reach a lower-level service, then it is safe to use `EFI_OPEN_PROTOCOL_GET_PROTOCOL` because the driver is informed through the layers if the lower-level protocol is removed.

The best example of this case in the EDK II is the FAT driver. The FAT driver uses the services of the Disk I/O Protocol to access the contents of a mass storage device. 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 `EFI_OPEN_PROTOCOL_BY_DRIVER`, so the FAT driver is also not allowed to open the Block I/O Protocol `EFI_OPEN_PROTOCOL_BY_DRIVER`. Instead, the FAT driver must use `EFI_OPEN_PROTOCOL_GET_PROTOCOL`. This method is safe because the FAT driver is 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.

5.1.3.4.3 Using `EFI_OPEN_PROTOCOL_BY_DRIVER`

The code fragment in shows the same example as above, but it retrieves the PCI I/O Protocol using the `EFI_OPEN_PROTOCOL_BY_DRIVER` attribute. When this attribute is used, the protocol must be closed when the UEFI Driver no longer requires the services of the PCI I/O Protocol. This example also shows `CloseProtocol()` being used to close the protocol, which is commonly found in implementations of `Supported()` and `Stop()`. Notice that the parameters passed to `CloseProtocol()` are identical to the parameters passed to `OpenProtocol()` with the `Interface` and `Attributes` parameters removed.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_DRIVER_BINDING_PROTOCOL *This;
EFI_HANDLE ControllerHandle;

//
// Retrieve PCI I/O Protocol interface on ControllerHandle
//
Status = gBS->OpenProtocol (
    ControllerHandle,           // Handle
    &gEfiPciIoProtocolGuid,    // Protocol
    NULL,                      // Interface
    This->DriverBindingHandle, // AgentHandle
    ControllerHandle,          // ControllerHandle
    EFI_OPEN_PROTOCOL_BY_DRIVER // Attributes
);
```

```

if (EFI_ERROR (Status)) {
    return Status;
}

//
// Close PCI I/O Protocol on ControllerHandle
//
Status = gBS->CloseProtocol (
    ControllerHandle,           // Handle
    &gEfiPciIoProtocolGuid,    // Protocol
    This->DriverBindingHandle, // AgentHandle
    ControllerHandle            // ControllerHandle
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 41—OpenProtocol() EFI_OPEN_PROTOCOL_BY_DRIVER

5.1.3.4.4 Using `EFI_OPEN_PROTOCOL_BY_DRIVER` | `EFI_OPEN_PROTOCOL_EXCLUSIVE`

The following code fragment shows the same example as above, but it retrieves the PCI I/O Protocol using both the `EFI_OPEN_PROTOCOL_BY_DRIVER` attribute and the `EFI_OPEN_PROTOCOL_EXCLUSIVE` attribute, which requests any other UEFI Driver that are using the PCI I/O Protocol release it.

There are only a very few instances where `EFI_OPEN_PROTOCOL_BY_DRIVER` | `EFI_OPEN_PROTOCOL_EXCLUSIVE` should be used. These are cases where a UEFI driver actually wants to gain exclusive access to a protocol, even if it requires stopping other UEFI drivers to do so.

This combination of attributes is used rarely. One example in the EDK II is the debug port driver that opens the Serial I/O Protocol with the `EFI_OPEN_PROTOCOL_BY_DRIVER` | `EFI_OPEN_PROTOCOL_EXCLUSIVE` attribute. This attribute allows a debugger to take control of a serial port even if it is already being used as a console device. *If this device is the only console device in the system, then the user loses the only console device when the debug port driver is started.*

Caution: This open mode can be **dangerous** if the system requires the services produced by the UEFI drivers that are stopped.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/PciIo.h>

EFI_STATUS           Status;
EFI_DRIVER_BINDING_PROTOCOL *This;
EFI_HANDLE           ControllerHandle;

//
// Retrieve PCI I/O Protocol interface on ControllerHandle
//
Status = gBS->OpenProtocol (
    ControllerHandle,           // Handle

```

```

        &gEfiPciIoProtocolGuid,           // Protocol
        NULL,                          // Interface
        This->DriverBindingHandle,     // AgentHandle
        ControllerHandle,              // ControllerHandle
        EFI_OPEN_PROTOCOL_BY_DRIVER | // Attributes
        EFI_OPEN_PROTOCOL_EXCLUSIVE
    );
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Close PCI I/O Protocol on ControllerHandle
//
Status = gBS->CloseProtocol (
    ControllerHandle,              // Handle
    &gEfiPciIoProtocolGuid,        // Protocol
    This->DriverBindingHandle,    // AgentHandle
    ControllerHandle              // ControllerHandle
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

**Example 42—OpenProtocol() EFI_OPEN_PROTOCOL_BY_DRIVER |
EFI_OPEN_PROTOCOL_EXCLUSIVE**

5.1.3.4.5 Using `EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER`

The code fragment below shows an example that may be used by a UEFI Bus Driver that produces child handles. This specific example shows the PCI bus driver creating a child handle, opening the PCI Root Bridge I/O Protocol using the `EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER` attribute on behalf of a child PCI controller that the PCI bus driver created, closing the PCI Root Bridge I/O Protocol, and destroying the child handle. These operations are typically spread across the Start() and Stop() functions.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/PciRootBridgeIo.h>
#include <Protocol/DevicePath.h>
#include <Protocol/PciIo.h>

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

//
// Create new child handle
//
ChildHandle = NULL;
Status = gBS->InstallMultipleProtocolInterfaces (

```

```

        &ChildHandle,
        &gEfiDevicePathProtocolGuid, DevicePath,
        &gEfiPciIoProtocolGuid,      PciIo,
        NULL
    );
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Open parent PCI Root Bridge I/O Protocol
//
Status = gBS->OpenProtocol (
    ControllerHandle,                               //Handle
    &gEfiPciRootBridgeIoProtocolGuid,             //Protocol
    (VOID **)&PciRootBridgeIo,                  //Interface
    This->DriverBindingHandle,                   //AgentHandle
    ChildHandle,                                 //ControllerHandle
    EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER //Attributes
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Close parent PCI Root Bridge I/O Protocol
//
Status = gBS->CloseProtocol (
    ControllerHandle,                           // Handle
    &gEfiPciRootBridgeIoProtocolGuid, // Protocol
    This->DriverBindingHandle,           // AgentHandle
    ChildHandle,                         // ControllerHandle
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Destroy child handle
//
Status = gBS->UninstallMultipleProtocolInterfaces (
    ChildHandle,
    &gEfiDevicePathProtocolGuid, DevicePath,
    &gEfiPciIoProtocolGuid,      PciIo,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

**Example 43—OpenProtocol()
EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER**

5.1.3.5 OpenProtocolInformation()

This service retrieves the list of agents currently using a specific protocol interface installed on a handle in the handle database. An agent may be a UEFI Driver or a UEFI

Application using the services of a protocol interface. The `OpenProtocol()` service adds agents to the list, and the `CloseProtocol()` service removes agents from the list. The return buffer from this service is allocated using `AllocatePool()`. To prevent memory leaks, the caller must free the return buffer with `FreePool()` when it no longer needs it.

The UEFI Shell command `openinfo` uses this service to view the results from `OpenProtocolInformation()` for any protocol installed into the handle database. It is very useful when debugging UEFI Drivers to evaluate the state of protocols the drivers consume and produce in the handle database and to verify that the UEFI Driver is using `OpenProtocol()` and `CloseProtocol()` properly.

A UEFI Driver may use this service to find the list of controllers the UEFI Driver is managing or the list of child handles that the UEFI driver has produced in previous calls to the `Start()`. A UEFI Driver may also choose to keep track of this type of information itself and not use the Protocol Handler Services to retrieve this type of information.

The following code fragment uses `LocateHandleBuffer()` to retrieve the list of handles that support the PCI Root Bridge I/O Protocol. It then uses `OpenProtocolInformation()` on the first handle that supports the PCI Root Bridge I/O Protocol to retrieve information on all the agents that are using that specific PCI Root Bridge I/O Protocol instance. This example then loops through all the consumers of that PCI Root Bridge I/O Protocol and counts the number of handles that have opened the PCI Root Bridge I/O Protocol instance with an open mode of `EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER`. This open mode indicates that the agent is a child handle. The result is the total number of PCI controllers that are attached to that specific PCI Root Bridge I/O Protocol instance.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>
#include <Protocol/PciRootBridgeIo.h>

EFI_STATUS Status;
UINTN HandleCount;
*HandleBuffer;
EFI_HANDLE ControllerHandle;
*OpenInfo;
EntryCount;
Index;
Attributes;
NumberOfChildren;

// 
// Retrieve array of handles that support the USB I/O Protocol
//
Status = gBS->LocateHandleBuffer (
    ByProtocol,
    &gEfiPciRootBridgeIoProtocolGuid,
    NULL,
    &HandleCount,
    &HandleBuffer
);
if (EFI_ERROR (Status)) {
    return Status;
}
if (HandleCount == 0) {
    return EFI_NOT_FOUND;
}
```

```

}

// Assign ControllerHandle to the first handle in the array
//
ControllerHandle = HandleBuffer[0];

//
// Free the array of handles
//
FreePool (HandleBuffer);

//
// Retrieve information about how the PCI Root Bridge I/O Protocol
// instance on ControllerHandle is being used.
//
Status = gBS->OpenProtocolInformation (
    ControllerHandle,
    &gEfiPciRootBridgeIoProtocolGuid,
    &OpenInfo,
    &EntryCount
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Count the number child handles that are currently using the PCI Root
// Bridge I/O Protocol on ControllerHandle children
//
for (Index = 0, NumberOfChildren = 0; Index < EntryCount; Index++) {
    Attributes = OpenInfo[Index].Attributes;
    if ((Attributes & EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER) != 0) {
        NumberOfChildren++;
    }
}

//
// Free the buffer allocated by OpenProtocolInformation()
//
FreePool (OpenInfo);

```

Example 44—Count child handles using OpenProtocolInformation()

5.1.4 Task Priority Level(TPL) Services

The Task Priority Level Services provide a mechanism for code to execute code at a raised priority for short periods of time. One use case is a UEFI Driver that is required to raise the priority because the implementation of a service of a specific protocol requires execution at a specific TPL to be UEFI conformant. Another use case is a UEFI Driver that needs to implement a simple lock, or critical section, on global data structures maintained by the UEFI Driver. Event notification functions, covered in the next section, always execute at raised priority levels.

The service `RaiseTPL()` is used to raise the priority level from its current level to a higher level and return the priority level before it was raised. The service `RestoreTPL()`

is used to restore a the priority level to a priority level returned by `RaiseTPL()`. These two services are always used in pairs.

Note: *There are no UEFI services provided to lower the TPL, and it is illegal to use `RaiseTPL()` to attempt to raise the priority level to a level below the current priority level. If attempted, the behavior of the platform is indeterminate.*

The Event, Timer, and Task Priority Services section of the *UEFI Specification* defines four TPL levels. These are `TPL_APPLICATION`, `TPL_CALLBACK`, `TPL_NOTIFY`, and `TPL_HIGH_LEVEL`. UEFI Driver and UEFI Applications are started at `TPL_APPLICATION`. UEFI Drivers should execute code at the lowest possible TPL level and minimize the time spent at raised TPL levels.

Note: *Only `TPL_APPLICATION`, `TPL_CALLBACK`, `TPL_NOTIFY`, and `TPL_HIGH_LEVEL` may be used by UEFI Drivers. All other values are reserved for use by the firmware. Using them results in unpredictable behavior. Good coding practice dictates that all code should execute at its lowest possible TPL level, and the use of TPL levels above `TPL_APPLICATION` must be minimized. Executing at TPL levels above `TPL_APPLICATION` for extended periods of time may also result in unpredictable behavior.*

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 that are accessed from both normal context and interrupt context must be protected by a lock.

The following code fragment shows how the `RaiseTPL()` and `RestoreTPL()` 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.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

UINT32 gCounter;

EFI_TPL OldTpl;

//
// Raise the Task Priority Level to TPL_HIGH_LEVEL to block timer
// interrupts
//
OldTpl = gBS->RaiseTPL (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 45—Using TPL Services for a Global Lock

The code fragment in Example 46, below, has the same functionality as [Example 45](#), above, but uses the lock macros and functions from the EDK II Library `UefiLib` that use `RaiseTPL()` and `RestoreTPL()` to implement general purpose locks. The global variable `gLock` is an `EFI_LOCK` structure that is initialized using the `EFI_INITIALIZE_LOCK_VARIABLE()` macro that specifies the use of `TPL_HIGH_LEVEL` when the lock is acquired. The `EfiAcquireLock()` and `EfiReleaseLock()` functions hide the details of managing TPL levels.

```
#include <Uefi.h>
#include <Library/UefiLib.h>

EFI_LOCK gLock = EFI_INITIALIZE_LOCK_VARIABLE (TPL_HIGH_LEVEL);

UINT32 gCounter;

//
// Acquire the lock to block timer interrupts
//
EfiAcquireLock (&gLock);

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

//
// Release the lock
//
EfiReleaseLock (&gLock);
```

Example 46—Using UEFI Library for a Global Lock

The algorithm shown in these two global lock examples also applies to a UEFI Driver that is required to implement protocol services that execute at a specific TPL level. For example, the services in the Block I/O Protocol must be called at or below `TPL_CALLBACK`. This means that the implementation of the `ReadBlocks()`, `WriteBlocks()`, and `FlushBlocks()` services should raise the priority level to `TPL_CALLBACK`. This would be identical to Example 46, above, but would use `TPL_CALLBACK` instead of `TPL_HIGH_LEVEL`.

5.1.5 Event services

UEFI Boot Services are provided to create, manage, and close UEFI Events. UEFI Drivers may use these event services for several features that may include the following:

- Implementation of protocols that produce an `EFI_EVENT` to inform protocol consumers when input is available.
- Notification when `ExitBootServices()` is called by an OS Loader or OS Kernel so UEFI Drivers can place devices in a quiescent state or a state that is required for OS compatibility.
- Notification when `SetVirtualAddressMap()` is called by an OS Loader or OS Kernel so a UEFI Runtime Driver can translate physical addresses to virtual addresses.

- Timer events used to periodically poll for I/O completion and/or detect timeout conditions.
- Implementation of protocols that provide non-blocking I/O capabilities where notification of an I/O completion utilizes an `EFI_EVENT`.

5.1.5.1 `CreateEvent()`, `CreateEventEx()`, and `CloseEvent()`

The `CreateEvent()`, `CreateEventEx()`, and `CloseEvent()` services are used to create and close 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 is invoked when an event is placed into the signaled state with a call to `SignalEvent()`. The notification function for wait type events is invoked when the event is passed to the `CheckEvent()` or `WaitForEvent()` services.

UEFI Drivers that produce protocols providing an `EFI_EVENT` field to indicate when input is available are required to create events of type `EVT_NOTIFY_WAIT`. Consumers of these protocols may use `CheckEvent()` or `WaitForEvent()` to check when input is available. Protocols from the *UEFI Specification* containing this use case include the Simple Text Input Protocols, the Pointer Protocols, and the Simple Network Protocol. The complete list follows:

- `EFI_ABSOLUTE_POINTER_PROTOCOL`
- `EFI_SIMPLE_NETWORK_PROTOCOL`
- `EFI_SIMPLE_POINTER_PROTOCOL`
- `EFI_SIMPLE_TEXT_INPUT_PROTOCOL`
- `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL`

Some UEFI drivers are required to place their controllers in a quiescent state or perform other controller-specific actions when 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 `ExitBootServices()` is called by the operating system.

UEFI Runtime Drivers may need to be notified when `SetVirtualAddressMap()` is called to convert physical addresses to virtual addresses. A complete example for this use case, including the use of `CreateEventEx()`, is shown in [Section 5.2.9](#).

UEFI Drivers may use timer events to periodically poll for device status changes, poll for an I/O completion or detect timeouts. A complete example showing how to create periodic and one-shot timer events using `CreateEventEx()` is provided in [Section 5.1.6](#).

Note: If a UEFI Driver creates events in its driver entry point, those events must be closed with `CloseEvent()` in the UEFI Driver's `Unload()` function.

Note: If a UEFI Driver creates events in its Driver Binding Protocol `Start()` function associated with a device, those events must be closed with `CloseEvent()` in its Driver Binding Protocol `Stop()` function.

Note: If a UEFI Driver creates events as part of an I/O operation, the event should be closed with `CloseEvent()` when the I/O operation is completed.

Caution: If the `CloseEvent()` service is not used to close events created with `CreateEvent()` or `CreateEventEx()`, the event consumes memory and generates a memory leak.

The code fragment below shows an example of a wait event created by a keyboard driver producing the `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL`. The first part of the code fragment is the event notification function plus an internal worker function that are called when the status of the wait event is checked with the `CheckEvent()` or the `WaitForEvent()` services. The second part of the code fragment is the code from the Driver Binding Protocol `start()` and `stop()` functions that create and close the wait event. Typically, a UEFI application or the UEFI boot manager call `CheckEvent()` or `WaitForEvent()` to see if a key has been pressed on a input device that supports the Simple Text Input Ex Protocol. This call to `CheckEvent()` or `WaitForEvent()` causes the notification function of the wait event in the Simple Text Input Ex 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, the wait event is signaled with a call to `SignalEvent()`. If the wait event is signaled, the UEFI application or UEFI boot manager then receives an `EFI_SUCCESS` return code and the UEFI application or UEFI boot manager calls the `ReadKeyStroke()` service of the `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` to read the key that was pressed.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/SimpleTextInEx.h>

EFI_STATUS
KeyboardCheckForKey (
  VOID
)
{
  //
  // Perform hardware specific action to detect if a key on a
  // keyboard has been pressed.
  //
  return EFI_SUCCESS;
}

VOID
EFI API
NotifyKeyboardCheckForKey (
  IN EFI_EVENT Event,
  IN VOID      *Context
)
{
  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL  *SimpleInputEx;

  SimpleInputEx = (EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *)Context;
  if (!EFI_ERROR (KeyboardCheckForKey ())) {
    gBS->SignalEvent (SimpleInputEx->WaitForKeyEx);
  }
}

EFI_STATUS
Status;
EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL  *SimpleInputEx;
```

```

// Create a wait event for a Simple Input Protocol
//
Status = gBS->CreateEvent (
    EVT_NOTIFY_WAIT,           // Type
    TPL_NOTIFY,                // NotifyTpl
    NotifyKeyboardCheckForKey, // NotifyFunction
    SimpleInputEx,             // NotifyContext
    &(SimpleInputEx->WaitForKeyEx) // Event
);
if (EFI_ERROR (status)) {
    return Status;
}

//
// Close the wait event
//
Status = gBS->CloseEvent (SimpleInputEx->WaitForKeyEx);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 47—Create and close a wait event

The code fragment in the following example shows how an Exit Boot Services event is created using `CreateEvent()` and closed using `CloseEvent()`. In this example, the `EFI_EVENT` is a global variable. This is the typical implementation for a UEFI Driver because events of this type are usually created in the Driver Binding Protocol `start()` function and closed in the Driver Binding Protocol `stop()` function, and the global variable provides an easy method to close the event in the Driver Binding Protocol `Stop()` function.

This example also contains the function `NotifyExitBootService()`, a template for the event notification function. It should contain the set of UEFI Driver specific actions that must be performed when the OS Load or OS Kernel calls `ExitBootServices()`. This notification function is registered in the call to `CreateEvent()`. The execution priority level is `TPL_NOTIFY` and the `NotifyContext` is `NULL` in this example.

Caution: *The notification function for `ExitBootServices()` is not allowed to use any of the UEFI Memory Services, either directly or indirectly, because using those services may modify the UEFI Memory Map and force an error to be returned from `ExitBootServices()`. An OS loader or OS Kernel that calls `ExitBootServices()` needs to know the state of the memory map at the time `ExitBootServices()` was called. The OS loader retrieves the current state of the memory map by calling `GetMemoryMap()`. If events registered on `ExitBootServices()` perform memory allocation or free calls, the memory map may be modified, and may cause incorrect memory map information to be used by the OS. The UEFI memory manager detects when the memory map is modified, so the OS loader always knows that the memory map was not modified if `ExitBootServices()` returns `EFI_SUCCESS`. If the memory map was modified, the OS loader must call `GetMemoryMap()` again to get the current memory map state, and then retry a call to `ExitBootServices()`. The modified state is cleared during the call to `GetMemoryMap()`.*

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
```

```

// Global variable for Exit Boot Services event
//
EFI_EVENT mExitBootServicesEvent = NULL;

VOID
EFIAPI
NotifyExitBootServices (
    IN EFI_EVENT Event,
    IN VOID        *Context
)
{
//
// Put driver-specific actions here to place controllers into
// an idle state. No UEFI Memory Service may be used directly
// or indirectly.
//
}

EFI_STATUS Status;

//
// Create an Exit Boot Services event.
//
Status = gBS->CreateEvent (
    EVT_SIGNAL_EXIT_BOOT_SERVICES,           // Type
    TPL_NOTIFY,                            // NotifyTpl
    NotifyExitBootServices,                // NotifyFunction
    NULL,                                  // NotifyContext
    &mExitBootServicesEvent               // Event
);
if (EFI_ERROR (Status)) {
    return Status;
}

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

```

Example 48—Create and Close an Exit Boot Services Event

The following [code fragment](#) has the same functionality as Example 48, above, but uses `CreateEventEx()` instead of `CreateEvent()` to create an event that is signaled when `ExitBootServices()` is called. `CreateEventEx()` supports event groups that are named by GUID. The Event, Timer, and Task Priority Services section of the *UEFI Specification* defines a set of event group GUIDs that are defined in the EDK II in the `MdePkg` include file `<Guid/EventGuid.h>`.

Caution: `CreateEventEx()` allows creation of more than one timer event associated with the same event group GUID. Because there is no mechanism for determining which of the timer events associated with the same event group GUID was signaled, it is recommended that timer events be created with `CreateEvent()` or with `CreateEventEx()` using a NULL EventGroup.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Guid/EventGroup.h>

//
// Global variable for Exit Boot Services event
//
EFI_EVENT mExitBootServicesEvent = NULL;

VOID
EFIAPI
NotifyExitBootServices (
    IN EFI_EVENT Event,
    IN VOID        *Context
)
{
//
// Put driver-specific actions here to place controllers into
// an idle state. No UEFI Memory Service may be used directly
// or indirectly.
//
}

EFI_STATUS Status;

//
// Create an Exit Boot Services event using event group GUID.
//
Status = gBS->CreateEventEx (
    EVT_NOTIFY_SIGNAL,           // Type
    TPL_NOTIFY,                 // NotifyTpl
    NotifyExitBootServices,     // NotifyFunction
    NULL,                       // NotifyContext
    &gEfiEventExitBootServicesGuid, // EventGroup
    &mExitBootServicesEvent      // Event
);

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

```

Example 49—Create and Close an Exit Boot Services Event Group

[Example 49](#), above, shows how the `CreateEventEx()` function is used to create an event that is notified when an event group named by GUID is signaled. In this case, notification functions are called when the OS Loader or OS Kernel calls `ExitBootServices()`. `CreateEventEx()` also supports creating an event for an event group named by GUID that causes all the event notification functions associated with that same event group to be executed when the event is signaled with `SignalEvent()`.

The [example below](#) shows the simplest method of creating, signaling, and closing an event group named by `gEfiExampleEventGroupGuid`. Notice that Type is 0 and no notification function, TPL, or context is specified. Since use of this mechanism is usually

in cases where one UEFI image needs to signal events in other UEFI images, this specific usage of `CreateEventEx()` is rarely used by UEFI Drivers.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Guid/ExampleEventGroup.h>

EFI_STATUS Status;
EFI_EVENT Event;

//
// Create event that is used to signal an event group
//
Status = gBS->CreateEventEx (
    0,                                // Type
    0,                                // NotifyTpl
    NULL,                             // NotifyFunction
    NULL,                             // NotifyContext
    &gEfiExampleEventGroupGuid,        // EventGroup
    &Event                            // Event
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Signal the event causing all notification functions for this
// event group to be executed
//
Status = gBS->SignalEvent (Event);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Close the event
//
Status = gBS->CloseEvent (Event);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 50—Create and Signal an Event Group

5.1.5.2 `SignalEvent()`

This service places an event in the signaled state. Use `SignalEvent()` in implementations of protocols containing an `EFI_EVENT` field informing a consumer of the protocol when input is ready. The protocols from the *UEFI Specification* containing this use case include the Simple Text Input Protocols, the Pointer Protocols, and the Simple Network Protocol. The complete list follows:

- `EFI_ABSOLUTE_POINTER_PROTOCOL`
- `EFI_SIMPLE_NETWORK_PROTOCOL`

- `EFI_SIMPLE_POINTER_PROTOCOL`
- `EFI_SIMPLE_TEXT_INPUT_PROTOCOL`
- `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL`

The example below shows the Simple Text Input Ex Protocol that signals the `EFI_EVENT` in that protocol when a key press has been detected. The function `KeyboardCheckForKey()` is a hardware specific function that returns `EFI_SUCCESS` if a key has been pressed. It returns an error code if a key has not been pressed. The check is performed at `TPL_NOTIFY` to guarantee that hardware action checking for a key press is atomic.

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>
#include <Protocol/SimpleTextInEx.h>

EFI_STATUS
EFIAPI
KeyboardCheckForKey (
  VOID
)
{
  //
  // Perform hardware specific action to detect if a key on a
  // keyboard has been pressed.
  //
  return EFI_SUCCESS;
}

EFI_STATUS Status;
EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *SimpleInputEx;
EFI_TPL OldTpl;

//
// Enter critical section
//
OldTpl = gBS->RaiseTPL (TPL_NOTIFY);

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

//
// Leave critical section
//
gBS->RestoreTPL (OldTpl);
```

Example 51—Signal a key press event

`SignalEvent()` is also used by UEFI Drivers required to signal an event associated with the completion of a non-blocking I/O operation. The protocols in the *UEFI Specification* containing this use-case include the Network Protocols, SCSI Protocols, ATA Protocols, and the Block I/O 2 Protocol. The complete list follows:

- `EFI_ARP_PROTOCOL`
- `EFI_IPSEC_PROTOCOL`
- `EFI_IPSEC2_PROTOCOL`
- `EFI_IPSEC_CONFIG_PROTOCOL`
- `EFI_MANAGED_NETWORK_PROTOCOL`
- `EFI_ATA_PASS_THRU_PROTOCOL`
- `EFI_BLOCK_IO2_PROTOCOL`
- `EFI_SCSI_IO_PROTOCOL`
- `EFI_EXT_SCSI_PASS_THRU_PROTOCOL`
- `EFI_DHCP4_PROTOCOL`
- `EFI_IP4_PROTOCOL`
- `EFI_IP4_CONFIG_PROTOCOL`
- `EFI_MTFTP4_PROTOCOL`
- `EFI_TCP4_PROTOCOL`
- `EFI_UDP4_PROTOCOL`
- `EFI_FTP4_PROTOCOL`
- `EFI_DHCP6_PROTOCOL`
- `EFI_IP6_PROTOCOL`
- `EFI_IP6_CONFIG_PROTOCOL`
- `EFI_MTFTP6_PROTOCOL`
- `EFI_TCP6_PROTOCOL`
- `EFI_UDP6_PROTOCOL`

5.1.5.3 CheckEvent()

This service checks to see if an event is in the waiting state or the signaled state. EFI Drivers creating events use this service to determine when an event has been signaled with `signalEvent()`. Such events include timer events, those used to determine when input is available, or events associated with non-blocking I/O operations.

The example below is an example that creates a one-shot timer event signaled 4 seconds in the future. `CheckEvent()` is called in a loop waiting for the timer event to be signaled.

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>

EFI_STATUS Status;
EFI_EVENT TimerEvent;

Status = gBS->CreateEvent (
    EVT_TIMER | EVT_NOTIFY_WAIT, // Type
    TPL_NOTIFY,                // NotifyTpl
    NULL,                      // NotifyFunction
    NULL,                      // NotifyContext
    &TimerEvent                // Event
);
if (EFI_ERROR (Status)) {
    return Status;
}

Status = gBS->SetTimer (
    TimerEvent,
    TimerRelative,
    EFI_TIMER_PERIOD_SECONDS (4)
);
if (EFI_ERROR (Status)) {
    gBS->CloseEvent (TimerEvent);
    return Status;
}

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

Example 52—Wait for one-shot timer event to be signaled

5.1.6 SetTimer()

This service programs a timer event to be signaled in the future. The time is specified in 100 nanosecond (ns) units. UEFI supports both periodic timer events and one-shot timer events. Use these timer events when polling for I/O completions, detecting hot plug events, detecting timeout conditions for I/O operations, supporting asynchronous I/O operations, etc.

Caution: *The units used for timer events may appear to have better accuracy than the `Stall()` service, which has an accuracy of 1 µs, but that may not be the case. UEFI uses a*

single timer interrupt to determine when to signal timer events. The resolution of timer events is dependent on the frequency of the timer interrupt.

UEFI system firmware uses a hardware timer interrupt to measure time. These time measurements are used to determine when enough time has passed to signal a timer event programmed with `SetTimer()`. In most systems, the timer interrupt is generated every 10 ms to 50 ms, but the *UEFI Specification* does not require any specific interrupt rate. This lack of specificity means that a periodic timer programmed with a period much smaller than 10 ms may only be signaled every 10 ms to 50 ms. If short delays much smaller than 10 ms are required, use the `Stall()` service.

TIP: Timer event services are not accurate over short delays. If a short, accurate delay, is required then the `Stall()` service should be used.

The code fragment in [Example 53](#) shows how to create a timer event and program it as a periodic timer with a period of 100 ms. When the created event is signaled every 100 ms, the notification function `TimerHandler()` is called at `TPL_NOTIFY` with the `EXAMPLE_DEVICE` context that was registered when the event was created. The EDK II library `UefiLib` provides macros for the timer periods used with the `SetTimer()` services. These macros include `EFI_TIMER_PERIOD_MICROSECONDS()`, `EFI_TIMER_PERIOD_MILLISECONDS()`, and `EFI_TIMER_PERIOD_SECONDS()`.

The Private Context Structure a UEFI Driver uses to store device specific information usually contains `EFI_EVENT` fields for the events the UEFI Driver creates. This allows a UEFI Driver to close events when a device is stopped or when a UEFI Driver is unloaded. In this example, the Private Context Structure called `EXAMPLE_DEVICE` contains an `EFI_EVENT` for both a periodic and a one-shot timer. The Private Context Structure is also typically passed in as the `Context` parameter when an event is created. This provides the event notification function with the device specific context required to perform the device specific actions.

Caution: Always close timer events with the UEFI Boot Service `CloseEvent()` whenever a device is stopped or a UEFI Driver is unloaded. If not performed, a call for an event notification no longer present in memory, or event notification function for a device no longer available, may cause unexpected failures.

```

#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>
#include <Library/UefiLib.h>

typedef struct {
    UINTN     Signature;
    EFI_EVENT PeriodicTimer;
    EFI_EVENT OneShotTimer;
    //
    // Other device specific fields
    //
} EXAMPLE_DEVICE;

VOID
TimerHandler (
    IN EFI_EVENT Event,
    IN VOID      *Context
)
{
    //
    // Perform a UEFI driver-specific operation.
    //
}

EFI_STATUS
Status;
EXAMPLE_DEVICE *Device;

Status = gBS->CreateEvent (
    EVT_TIMER | EVT_NOTIFY_SIGNAL,    // Type
    TPL_NOTIFY,                     // NotifyTpl
    TimerHandler,                  // NotifyFunction
    Device,                         // NotifyContext
    &Device->PeriodicTimer        // Event
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Program the timer event to be signaled every 100 ms.
//
Status = gBS->SetTimer (
    Device->PeriodicTimer,
    TimerPeriodic,
    EFI_TIMER_PERIOD_MILLISECONDS (100)
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 53—Create periodic timer event

The following code fragment shows how to create a one-shot timer event that is signaled 4 seconds in the future. When the created event is signaled, the notification function `TimerHandler()` is called at `TPL_CALLBACK` with the `EXAMPLE_DEVICE` context that was registered when the event was created.

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>
#include <Library/UefiLib.h>

typedef struct {
    UINTN     Signature;
    EFI_EVENT PeriodicTimer;
    EFI_EVENT OneShotTimer;
    //
    // Other device specific fields
    //
} EXAMPLE_DEVICE;

VOID
TimerHandler (
    IN EFI_EVENT Event,
    IN VOID      *Context
)
{
    //
    // Perform a UEFI driver-specific operation.
    //
}

EFI_STATUS
Status;
EXAMPLE_DEVICE *Device;

Status = gBS->CreateEvent (
    EVT_TIMER | EVT_NOTIFY_SIGNAL, // Type
    TPL_CALLBACK,                // NotifyTpl
    TimerHandler,                // NotifyFunction
    Device,                      // NotifyContext
    &Device->OneShotTimer       // Event
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Program the timer event to be signaled 4 seconds from now.
//
Status = gBS->SetTimer (
    Device->OneShotTimer,
    TimerRelative,
    EFI_TIMER_PERIOD_SECONDS (4)
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 54—Create one-shot timer event

The code fragment below shows how to cancel and close the one-shot timer created in Example 54 above. If the UEFI Driver completes an I/O operation normally, any timer events used to detect timeout conditions must be canceled. If the timeout condition is only used as part of device detection, the timer event may not be required again. In those cases, the event can be both canceled and closed.

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>
#include <Library/UefiLib.h>

typedef struct {
    UINTN     Signature;
    EFI_EVENT PeriodicTimer;
    EFI_EVENT OneShotTimer;
    //
    // Other device specific fields
    //
} EXAMPLE_DEVICE;

EFI_STATUS      Status;
EXAMPLE_DEVICE *Device;

//
// Cancel the one-shot timer event.
//
Status = gBS->SetTimer (Device->OneShotTimer, TimerCancel, 0);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Close the one-shot timer event.
//
Status = gBS->CloseEvent (Device->OneShotTimer);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 55—Cancel and close one-shot timer event

5.1.7 Stall()

The **Stall()** service waits for a specified number of microseconds. In 32-bit environments, the range of supported delays is from 1 μ s to a little over an hour. In 64-bit execution environments, the range of supported delays is from 1uS to about 500,000 years. However, the delays passed into this service should be short and are typically in the range of a few microseconds to a few milliseconds.

Caution: *Implementations of the **Stall()** service may disable interrupts and may block execution of other UEFI drivers. If long delays are required, use a Timer Event instead. See **CreateEvent()**, **CreateEventEx()**, and **SetTimer()** for details.*

The **Stall()** service is very accurate and typically uses a high frequency hardware timer or a calibrated software delay loop to implement the stall functionality.

Caution: `Stall()` may use a different timing source than the event timer, and may have a higher or lower frequency and, hence, different accuracy.

For hardware devices requiring delays between register accesses, use the `Stall()` service, with a fixed stall value based in a hardware specification for the device being accessed. The following example shows a use-case to perform a fixed delay of 10 us between two PCI MMIO register writes.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT8 Value;

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

//
// Wait 10 us
//
gBS->Stall (10);

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

Example 56—Fixed delay stall

In this example, a UEFI driver sends a command to a controller and then waits for the command to complete. Use the `Stall()` service inside a loop to periodically check for the completion status. The example below shows how to poll for a completion status every millisecond and timeout after 100 ms.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
UINTN TimeOut;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT8 Value;

//
// Loop waiting for the register at Offset 0 of Bar #0 of PciIo to
// become 0xE0. Wait 1 ms between each check of this register, and
// time out if it does not become 0xE0 after 100 ms.
//
for (TimeOut = 0; TimeOut <= 100000; TimeOut += 1000) {
    //
    // Do a single 8-bit MMIO read from BAR #0, Offset 0 into Value
    //
    Status = PciIo->Mem.Read (
        PciIo,           // This
        EfiPciIoWidthUint8, // Width
        0,               // BarIndex
        0,               // Offset
        1,               // Count
        &Value           // Buffer
    );
    if (!EFI_ERROR (Status) && Value == 0xE0) {
        return EFI_SUCCESS;
    }

    //
    // Wait 1 ms
    //
    gBS->Stall (1000);
}
return EFI_TIMEOUT;
```

Example 57—Poll for completion status using stalls

5.2 Services that UEFI drivers rarely use

[Table 19](#) lists UEFI services rarely used by UEFI drivers. The following sub-topics briefly describe each service, why they are rarely used, or the particular circumstance in which they are useful. The code examples show how the services are typically used by UEFI drivers and are grouped by Service Type.

Table 19—UEFI services that are rarely used by UEFI drivers

Service	Type	Service Type	Notes
ConnectController()	Boot	Protocol Handler	Uses a set of precedence rules to find the best set of drivers to manage a controller.
DisconnectController()	Boot	Protocol Handler	Informs a set of drivers to stop managing a controller.
ReinstallProtocolInterface()	Boot	Protocol Handler	Reinstalls a protocol interface on a device handle.
LocateDevicePath()	Boot	Protocol Handler	Locates a device handle supporting a specific protocol and having the closest matching device path. UEFI drivers should use the services on the <i>ControllerHandle</i> passed into the <code>Supported()</code> and <code>start()</code> functions of the driver's <code>EFI_DRIVER_BINDING_PROTOCOL</code> .
LoadImage()	Boot	Image	Used only by bus drivers that can load, start, and potentially unload UEFI drivers stored in other images in some other location on the child devices of the bus.
StartImage()	Boot	Image	Used only by bus drivers that can load, start, and potentially unload UEFI drivers stored in other images in some other location on the child devices of the bus.
GetVariable()	Runtime	Variable	Returns the value of a variable.
SetVariable()	Runtime	Variable	Sets the value of a variable.
QueryVariableInfo()	Runtime	Variable	Returns information about the EFI variables.
GetTime()	Runtime	Time-related	Returns the current time and date, and the time-keeping capabilities of the platform.
CalculateCrc32()	Boot	Miscellaneous	Maintains the checksum of the UEFI System Table, UEFI boot services table, and UEFI runtime services table.
ConvertPointer()	Runtime	Miscellaneous	Sometimes used by UEFI runtime drivers. This service should never be used by UEFI boot service drivers.

Service	Type	Service Type	Notes
InstallConfigurationTable()	Boot	Miscellaneous	Adds, updates, or removes a configuration table from the UEFI system table.
WaitForEvent()	Boot	Event	Stops execution until an event is signaled.
GetNextMonotonicCount()	Boot	Special	Provides a 64-bit monotonic counter that is guaranteed to increase.

5.2.1 **ConnectController()** and **DisconnectController()**

These services request UEFI Drivers to start or stop managing controllers in a platform. They are typically used by the UEFI Boot Manager to connect the devices required to boot an operating system. These services may also be used by a UEFI Boot Manager to connect all devices in the platform if the user chooses to enter platform setup. OS Loaders and OS Installers may also use these services to connect additional devices required to complete an OS boot or OS installation operation.

Additionally, UEFI applications, such as the UEFI Shell, may use these services to test the functionality of a UEFI Driver under test. The UEFI Shell commands using these services are **connect**, **disconnect**, and **reconnect**. A common test sequence a UEFI Driver developer may use to test the functionality of a new UEFI Driver is:

- Load the UEFI Driver.
- Connect the UEFI Driver.
- Test functionality of protocols produced by the UEFI Driver.
- Disconnect the UEFI Driver.
- Unload the UEFI Driver.
- Fix known issues with the UEFI Driver and repeat.

The use of **ConnectController()** and **DisconnectController()** in UEFI Driver implementations is less common and is usually restricted to UEFI Drivers managing hot-plug capable busses and unloadable UEFI Drivers.

5.2.1.1 Hot Plug Operations

To facilitate a hot-add operation on a hot-plug capable bus, use **ConnectController()** to connect UEFI Drivers to the hot-added device. Likewise, to facilitate a hot-remove operation on a hot-plug capable bus, use **DisconnectController()** to request that UEFI Drivers stop managing the removed device. 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 pre-boot environment is dependent on the platform requirements for each bus type.

The best example of the hot-plug this use case in the EDK II is the USB Bus Driver in **MdeModulePkg/Bus/Usb/UsbBusDxe**. The USB bus driver in the EDK II does not create any

child handles in its Driver Binding Protocol `Start()` function. Instead, it registers a periodic timer event.

When the timer period expires, the timer event's notification function is called and that notification function examines all USB root ports and USB hubs to see if any USB devices have been added or removed. If a USB device is added, a child handle is created with a Device Path Protocol and a USB I/O Protocol. `ConnectController()` is then called to allow USB device drivers to connect to the newly added USB device. If a USB device has been removed, `DisconnectController()` is called to stop the USB device drivers from managing the removed USB device.

The following code fragment shows how `ConnectController()` is used to perform a recursive connect operation in response to a hot-add operation.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS Status;
EFI_HANDLE ChildHandle;

//
// Recursively connect all drivers to the hot-added device
//
Status = gBS->ConnectController (ChildHandle, NULL, NULL, TRUE);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 58—Recursive connect in response to a hot-add operation

The code fragment below shows how `DisconnectController()` is used to perform a recursive disconnect operation in response to a hot-remove operation.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS Status;
EFI_HANDLE ChildHandle;

//
// Recursively disconnect all drivers from the hot-removed device
//
Status = gBS->DisconnectController (
    ChildHandle,
    NULL,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 59—Recursive disconnect in response to a hot-remove operation

5.2.1.2 Driver Unload Operations

Use the `DisconnectController()` service, from unloadable UEFI drivers, to disconnect the UEFI driver from the device(s) it is managing. The `DisconnectController()` service is called from the `Unload()` function that is registered in the Loaded Image Protocol for the UEFI Driver.

The following code fragment shows a simple algorithm that a UEFI Driver can use to disconnect the UEFI Driver from all the devices in the system that it is currently managing.

It first retrieves the list of all the handles in the handle database, then disconnects the UEFI driver from each of those handles.

A UEFI Driver could implement a more efficient algorithm if the UEFI Driver kept a list of the controller handles it manages. It could then call `DisconnectController()` for each of the controller handles in that list.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>

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

//
// Retrieve array of all handles in the handle database
//
Status = gBS->LocateHandleBuffer (
    AllHandles,
    NULL,
    NULL,
    &HandleCount,
    &HandleBuffer
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Disconnect the current driver from all handles in the handle database
//
for (Index = 0; Index < HandleCount; Index++) {
    Status = gBS->DisconnectController (
        HandleBuffer[Index],
        gImageHandle,
        NULL
    );
}

//
// Free the array of handles
//
FreePool (HandleBuffer);
```

Example 60—Disconnect a UEFI Driver from all handles

5.2.2 ReinstallProtocolInterface()

This service should be used only to indicate media change events and when a device path is modified or updated. Some examples of when this service must be used are:

- A UEFI Driver that produces the Block I/O Protocol for a removable media device when the media in a removable media device is changed (i.e. Floppy, CD, DVD).
- A UEFI Driver that produces the Serial I/O Protocol when the attributes are modified using `SetAttributes()`
- A UEFI Driver that produces the Simple Network Protocol when the MAC address of the network interface is modified using `StationAddress()`.

Internally, this service performs the following series of actions:

1. `UninstallProtocolInterface()`, which may cause `DisconnectController()` to be called
2. `InstallProtocolInterface()`
3. `ConnectController()` to allow controllers that had to release the protocol a chance to connect to it again

Caution: *This service may induce reentrancy if a driver makes a request that requires a UEFI Driver for a parent device to call `ReinstallProtocolInterface()`. In this case, the driver making the request may not realize that the request causes 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 `SetAttributes()`. 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 forces the terminal driver to be disconnected. The terminal driver then attempts to connect to the serial device again, but the baud rate is the one that the terminal driver expects, so the terminal driver does 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 that care must be taken in the design of drivers that use this type of protocol.

The following code fragments in [Example 61](#) show what a UEFI driver that produces the Block I/O Protocol should do when the media in a removable media device is changed. The exact same protocol is reinstalled onto the controller handle. The specific action that detects if the media is not included in this code fragment. The original Block I/O Media structure is copied so it can be compared with the Block I/O Media structure after the media change detection logic is executed. The Block I/O Protocol is reinstalled if the Media ID is different, if the size of blocks on the mass storage device has changed, if the number of blocks on the mass storage device has changed, if the present status has changed, or if the media has changed from read-only to read-write or vice versa.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/BaseMemoryLib.h>

EFI_STATUS Status;
EFI_HANDLE ControllerHandle;
EFI_BLOCK_IO_PROTOCOL *BlockIo;
EFI_BLOCK_IO_MEDIA OldMedia;

//
// Make a copy of the current Block I/O Media structure
//
CopyMem (&OldMedia, &(BlockIo->Media), sizeof (EFI_BLOCK_IO_MEDIA));

//
// Perform driver specific action(s) required to detect if the
// media has been changed and update Block I/O Media structure.
//

//
// Detect whether it is necessary to reinstall the Block I/O Protocol.
//
if ((BlockIo->Media->MediaId      != OldMedia.MediaId) ||
    (BlockIo->Media->MediaPresent != OldMedia.MediaPresent) ||
    (BlockIo->Media->ReadOnly     != OldMedia.ReadOnly) ||
    (BlockIo->Media->BlockSize    != OldMedia.BlockSize) ||
    (BlockIo->Media->LastBlock   != OldMedia.LastBlock) ) {

    Status = gBS->ReinstallProtocolInterface (
                    ControllerHandle,
                    &gEfiBlockIoProtocolGuid,
                    BlockIo,
                    BlockIo
                );
    if (EFI_ERROR (Status)) {
        return Status;
    }
}

```

Example 61—Reinstall Block I/O Protocol for media change

The code fragments below show the Device Path Protocol for a Serial I/O device being reinstalled because the serial communication parameters that are expressed in a UART Device Path Node have been modified in a call to the `SetAttributes()` service of the Serial I/O Protocol.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS Status;
EFI_HANDLE ControllerHandle;
EFI_DEVICE_PATH_PROTOCOL *DevicePath;

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

//
// Check to see if the UART parameters have been modified
// and update UART node of DevicePath
//

//
//
//
Status = gBS->ReinstallProtocolInterface (
    ControllerHandle,
    &gEfiDevicePathProtocolGuid,
    DevicePath,
    DevicePath
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 62—Reinstall Device Path Protocol for Serial I/O attributes change

5.2.3 LocateDevicePath()

This service locates a device handle that supports a specific protocol and has the closest matching device path. Although a rare requirement, it is useful when a UEFI Driver needs to find an I/O abstraction for one of its parent controllers.

Normally, a UEFI Driver uses the services on the `ControllerHandle` that is passed into the `Supported()` and `Start()` functions of the EFI driver's `EFI_DRIVER_BINDING_PROTOCOL`.

However, if a UEFI Driver does require the use of services from a parent controller, `LocateDevicePath()` can be used to find the handle of a parent controller.

For example, a PCI device driver normally uses the PCI I/O Protocol to manage a PCI controller. Hypothetically, if a PCI device driver required 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. Then the `gBS->OpenProtocol()` service can be used to retrieve the PCI Root Bridge I/O Protocol interface from that handle.

The code fragment below shows how a UEFI Driver for a PCI Controller can retrieve the PCI Root Bridge I/O Protocol of which the PCI controller is a child.

Caution: *This operation is provided only as an illustration and 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.*

[Section 18.4.2, Example 175](#), contains another example showing the recommended method for a PCI driver to access the resources of other PCI controllers on the same PCI adapter without using the PCI Root Bridge I/O Protocol.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS Status;
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,
    (VOID **)&DevicePath,
    gImageHandle,
    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,
    (VOID **)&PciRootBridgeIo,
    gImageHandle,
    ControllerHandle,
    EFI_OPEN_PROTOCOL_GET_PROTOCOL
);
if (EFI_ERROR (status)) {
    return Status;
}

```

Example 63—Locate Device Path

5.2.4 [LoadImage\(\)](#) and [StartImage\(\)](#)

Use [LoadImage\(\)](#) to load and relocate a UEFI Image into system memory, and prepare it for execution. Use [StartImage\(\)](#) to transfer control to a UEFI Image that was previously loaded into system memory using [LoadImage\(\)](#). These services are typically used by the UEFI Boot Manager when processing load options for UEFI Drivers, UEFI Applications, or UEFI OS Loaders. UEFI drivers do not typically need to load other UEFI Drivers and/or UEFI applications.

One exception is a bus driver for a bus type that provides a storage container for UEFI Drivers and/or UEFI Applications. A PCI Option ROM is an example of a container with those attributes. A PCI Bus Driver is required to discover any PCI Option ROM containers present on PCI Adapters. If a PCI Option ROM contains one or more UEFI Drivers that are compatible with the currently executing CPU, then the PCI Bus Driver is required to load and start those UEFI Drivers using the [LoadImage\(\)](#) and [StartImage\(\)](#) services. The EDK II PCI Bus Driver that performs this operation can be found in [MdeModulePkg/Bus/Pci/PciBusDxe](#).

Another exception is a UEFI Driver that needs to execute a UEFI Application for the purposes of extended diagnostics or to augment driver configuration. There are UEFI standard methods for a UEFI Driver to provide diagnostics and configuration through the use of the [EFI_DRIVER_DIAGNOSTICS2_PROTOCOL](#) and HII. If for some reason, a UEFI Driver requires diagnostics or configuration capabilities that cannot be expressed using these standard methods, a UEFI Driver could choose to execute a UEFI Application that provides those capabilities. In the case of a PCI Adapter, UEFI Applications could be stored in the PCI Option ROM container. The UEFI Driver would use the [LoadImage\(\)](#) and [startImage\(\)](#) services to load and execute those UEFI Applications from that container.

The following code fragment in [Example 64](#) shows an example of a UEFI Driver for a PCI controller that uses the [LoadImage\(\)](#) and [StartImage\(\)](#) service to load and execute a 32 KB UEFI Application that is stored 32 KB into the PCI Option ROM container associated with the PCI controller. [PciControllerHandle](#) is the [EFI_HANDLE](#) for the PCI Controller.

This example retrieves both the PCI I/O Protocol and the Device Path Protocol associated with [PciControllerHandle](#). The Device Path Protocol is used to construct a

proper device path for the UEFI Application stored in the PCI option ROM. Helper functions from the EDK II library `DevicePathLib` are used to fill in the contents of a new device path node for the UEFI Application stored in the PCI Option ROM and to append that device path node to the device path of the PCI controller. Use the PCI I/O Protocol to access the shadowed copy of the PCI Option ROM contents through the `RomImage` field. The shadowed copy of the PCI Option ROM was created when the PCI bus was enumerated and the PCI I/O Protocols were produced.

Note: *The use of a 32 KB offset and 32 KB length simplifies this example. An add-in adapter that stores UEFI Applications in a PCI Option ROM container would likely define vendor specific descriptors to determine the offset and size of one or more UEFI Applications.*

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/DevicePathLib.h>
#include <Protocol/DevicePath.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_HANDLE PciControllerHandle;
*PciIo;
*PciDevicePath;
OptionRomNode;
*PciOptionRomDevicePath;
NewImageHandle;

// Retrieve PCI I/O Protocol associated with PciControllerHandle
//
Status = gBS->OpenProtocol (
    PciControllerHandle,
    &gEfiPciIoProtocolGuid,
    (VOID **) &PciIo,
    gImageHandle,
    NULL,
    EFI_OPEN_PROTOCOL_GET_PROTOCOL
);
if (EFI_ERROR (Status)) {
    return Status;
}

// Retrieve Device Path Protocol associated with PciControllerHandle
//
Status = gBS->OpenProtocol (
    PciControllerHandle,
    &gEfiDevicePathProtocolGuid,
    (VOID **) &PciDevicePath,
    gImageHandle,
    NULL,
    EFI_OPEN_PROTOCOL_GET_PROTOCOL
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

```

// Create Device Path Protocol to UEFI Application in PCI Option ROM
//
OptionRomNode.Header.Type      = MEDIA_DEVICE_PATH;
OptionRomNode.Header.SubType   = MEDIA_RELATIVE_OFFSET_RANGE_DP;
SetDevicePathNodeLength (&OptionRomNode.Header, sizeof (OptionRomNode));
OptionRomNode.StartingOffset  = BASE_32KB;
OptionRomNode.EndingOffset    = BASE_64KB - 1;
PciOptionRomDevicePath = AppendDevicePathNode (
    PciDevicePath,
    &OptionRomNode.Header
);

//
// Load UEFI Image from PCI Option ROM container
//
Status = gBS->LoadImage (
    FALSE,
    gImageHandle,
    PciOptionRomDevicePath,
    (UINT8 * )(PciIo->RomImage) + SIZE_32KB,
    SIZE_32KB,
    &NewImageHandle
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Start UEFI Image from PCI Option ROM container
//
Status = gBS->StartImage (NewImageHandle, NULL, NULL);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 64—Load and Start a UEFI Application from a PCI Option ROM

5.2.5 GetVariable() and SetVariable()

Use `GetVariable()` and `SetVariable()` services to read and write UEFI variables. UEFI Drivers for add-in adapters, such as PCI adapters, should not use these services to access configuration information for the adapter. Instead, the add-in adapter should provide its own local storage for configuration information. UEFI Drivers provided with UEFI system firmware use UEFI variables to store configuration information. Examples found in the EDK II of UEFI Drivers use UEFI variables to store configuration information include the IPv4 and IPv6 network stacks in the `MdeModulePkg/Universal/Network` and the `NetworkPkg`.

Caution: Add-in cards should not store their configuration via variables. When the card is removed from the system, the variables related to its configuration become ownerless. There is no way to safely recover that data. In addition, it is impossible for the system designer to determine the amount of configuration data each card consumes. As such, there may simply not be enough space to store the configuration in a particular system's variable space. To ensure proper function, each card must store its own configuration on the add-in card.

A UEFI Variable is specified with a combination of a GUID and a Unicode string. The GUID prevents name collisions between different vendors. Each vendor may create GUIDs for their own storage and manage their own namespace of Unicode strings for the GUID they create. The Boot Manager chapter of the *UEFI Specification* defines the **EFI_GLOBAL_VARIABLE_GUID**, also known as **gEfiGlobalVariableGuid** in the EDK II, that is reserved for UEFI variables defined by the *UEFI Specification*. UEFI Drivers must never use this GUID to store their configuration information.

Caution: *UEFI Drivers must never use **EFI_GLOBAL_VARIABLE_GUID** or **gEfiGlobalVariableGuid** to store configuration information. This GUID is reserved for use by the UEFI Specification.*

When UEFI variables are stored, there are attributes that describe the visibility and persistence of the variable. The legal combinations of attributes include the following:

BOOTSERVICE_ACCESS

The variable is available for read and write access in the pre-boot environment before **ExitBootServices()** is called. The variable is not available after **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 pre-boot components.

BOOTSERVICE_ACCESS | RUNTIME_ACCESS

The variable is available for read and write access in the pre-boot environment before **ExitBootServices()** is called, and is available for read-only access from the OS runtime environment after **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 pre-boot 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 pre-boot environment before **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 pre-boot components.

NON_VOLATILE | BOOTSERVICE_ACCESS | RUNTIME_ACCESS

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

A UEFI Driver that is required to use UEFI variables to store configuration information typically accesses those UEFI variables in the implementation of the services provided by a **EFI_HII_CONFIG_ACCESS_PROTOCOL** protocol instance. The services **GetVariable()** and **SetVariable()** are used to get and set configuration information associated with HII setup screens provided by the UEFI Driver using the UEFI HII infrastructure that is described in more detail in [Chapter 12](#).

The attribute of **NON_VOLATILE | BOOTSERVICE_ACCESS | RUNTIME_ACCESS** is used to store configuration information that persists across resets and power cycles. It also allows for updates to this configuration information from operating systems that provide support for OS-present configuration changes using the HII database exported by the UEFI system firmware.

The attribute of `BOOTSERVICE_ACCESS` should be used with a UEFI variable used as a mailbox to store state information that is required by multiple HII forms or multiple HII callbacks.

The following code fragment shows how to write a configuration structure to a UEFI variable whose contents are preserved across resets and power cycles. The GUID value, GUID global variable, and the configuration structure associated with the GUID are all typically declared in a GUID include file in an EDK II package implemented by a vendor. The structure `EXAMPLE_CONFIGURATION` from `<Guid/ExampleConfigurationVariable.h>` is shown here in comments to provide additional context for this specific code fragment.

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>
#include <Guid/ExampleConfigurationVariable.h>

// Example configuration structure from ExampleConfigurationVariable.h
//typedef struct {
//  UINT32 Question1;
//  UINT16 Question2;
//  UINT8 Question3;
//} EXAMPLE_CONFIGURATION;

EFI_STATUS Status;
EXAMPLE_CONFIGURATION ExampleConfiguration;

Status = gRT->SetVariable (
    L"ExampleConfiguration", // VariableName
    &gEfiExampleConfigurationVariableGuid, // VendorGuid
    EFI_VARIABLE_NON_VOLATILE |
    EFI_VARIABLE_BOOTSERVICE_ACCESS |
    EFI_VARIABLE_RUNTIME_ACCESS, // Attributes
    sizeof (EXAMPLE_CONFIGURATION), // DataSize
    &ExampleConfiguration
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 65—Write configuration structure to a UEFI variable

The code fragment below shows how to use the `GetVariable()` service to read the configuration structure from the UEFI variable written in the previous example.

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>
#include <Guid/ExampleConfigurationVariable.h>

EFI_STATUS Status;
EXAMPLE_CONFIGURATION ExampleConfiguration;
UINTN DataSize;
UINT32 Attributes;

DataSize = sizeof (EXAMPLE_CONFIGURATION);
Attributes = EFI_VARIABLE_NON_VOLATILE |
              EFI_VARIABLE_BOOTSERVICE_ACCESS |
              EFI_VARIABLE_RUNTIME_ACCESS;
Status = gRT->GetVariable (
    L"ExampleConfiguration", // VariableName
    &gEfiExampleConfigurationVariableGuid, // VendorGuid
    &Attributes, // Attributes
    &DataSize, // DataSize
    &ExampleConfiguration // Data
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 66—Read configuration structure from a UEFI variable

The code fragment below is identical in functionality to the previous example, but uses the `GetVariable()` function from the EDK II library `UefiLib` to read the configuration structure from the UEFI variable. The UEFI variable contents are allocated from pool, so the variable contents must be freed after they are used. The `UefiLib` function `GetVariable()` supports reading both fixed size UEFI variables such as an `EXAMPLE_CONFIGURATION` structure and UEFI variables whose size may vary.

```
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Guid/ExampleConfigurationVariable.h>

EXAMPLE_CONFIGURATION *ExampleConfiguration;

ExampleConfiguration = GetVariable (
    L"ExampleConfiguration",
    &gEfiExampleConfigurationVariableGuid
);
if (ExampleConfiguration == NULL) {
    return EFI_NOT_FOUND;
}

// When done, free the UEFI variable contents
//
FreePool (ExampleConfiguration);
```

Example 67—Use UefiLib to read configuration structure from a UEFI variable

5.2.6 QueryVariableInfo()

Use this UEFI Runtime Service to retrieve information about the container used to store UEFI variables including their size, available space, and the maximum size of a single UEFI variable.

In general, UEFI Drivers do not use UEFI variables, and those UEFI Drivers that do use UEFI variables are provided with the UEFI system firmware where this type of information is usually already known. As a result, this service is rarely used by UEFI Drivers. It is more typically used by OS installers and OS kernels to determine the platform storage capabilities for UEFI variables.

The following code fragment shows how the `QueryVariableInfo()` service is used to collect information storage containers for UEFI variables that persist across reboots and power cycles and are available in both the pre-boot environment and by the OS.

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>

EFI_STATUS Status;
UINT64 MaximumVariableStorageSize;
UINT64 RemainingVariableStorageSize;
UINT64 MaximumVariableSize;

Status = gRT->QueryVariableInfo (
    EFI_VARIABLE_BOOTSERVICE_ACCESS | 
    EFI_VARIABLE_RUNTIME_ACCESS | 
    EFI_VARIABLE_NON_VOLATILE,
    &MaximumVariableStorageSize,
    &RemainingVariableStorageSize,
    &MaximumVariableSize
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 68—Collect information about the UEFI variable store

5.2.7 GetTime()

This service is rarely used. Use it only when the current time and date are required, such as marking the time and date of a critical error.

Caution: 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 `Stall()` service should be used for short delays. The `CreateEvent()`, `CreateEventEx()`, and `SetTimer()` services should be used for longer delays.

[Example 69](#) and [Example 70](#), following, are two examples of the `GetTime()` service. The first retrieves the current time and date in an `EFI_TIME` structure. The second retrieves both the current time and date in an `EFI_TIME` structure and the capabilities of the real-time clock hardware in an `EFI_TIME_CAPABILITIES` structure.

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>

EFI_STATUS Status;
EFI_TIME Time;

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

Example 69—Get time and date

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>

EFI_STATUS Status;
EFI_TIME Time;
EFI_TIME_CAPABILITIES Capabilities;

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

Example 70—Get real time clock capabilities

5.2.8 CalculateCrc32()

Use this service to maintain the checksums in the UEFI System Table, UEFI boot services table, and UEFI runtime services table. A UEFI driver that modifies one of these tables should use this service to update the checksums. A UEFI driver could compute the 32-bit CRC on its own, but the UEFI driver is smaller if it takes advantage of this UEFI boot service. This service can also be used to compute the checksums in Guided Partition Table(GPT) structures.

The following code fragment shows how `CalculateCrc32()` can be used to calculate and update the 32-bit CRC field in the UEFI System Table header. The EDK II library `UefiBootServicesTableLib` provides global variables for the UEFI System Table, the UEFI Boot Services Table, and the Image Handle for the currently executing driver. In this example, the global variable for the UEFI System Table called `gST` and the global variable for the UEFI Boot Services Table called `gBS` are used to reference the UEFI System Table header and call the UEFI Boot Services `CalculateCrc32()`. Since the CRC32 field is part of the structure for which the 32-bit CRC is being computed, it must be set to zero before calling `CalculateCrc32()`.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS Status;

gST->Hdr.CRC32 = 0;
Status = gBS->CalculateCrc32(
    &gST->Hdr,
    gST->Hdr.HeaderSize,
    &gST->Hdr.CRC32
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 71—Calculate and update 32-bit CRC in UEFI System Table

The code fragment below shows how to calculate a 32-bit CRC for an `EXAMPLE_DEVICE` structure. Since the computed 32-bit CRC is not stored within the `EXAMPLE_DEVICE` structure, it does not need to be zeroed before calling the `CalculateCrc32()` service.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS      Status;
EXAMPLE_DEVICE  Device;
UINT32          Crc;

Status = gBS->CalculateCrc32(&Device, sizeof (Device), &Crc);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 72—Calculate and 32-bit CRC for a structure

The `CalculateCrc32()` service can also be used to verify a 32-bit CRC value. The code fragment below shows how the 32-bit CRC for the UEFI System Table header can be verified. This algorithm preserves the original contents of the UEFI System Table header. It returns `TRUE` if the 32-bit CRC is good. Otherwise, it returns `FALSE`.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS      status;
UINT32          OriginalCrc32;
UINT32          Crc32;

OriginalCrc32  = gST->Hdr.CRC32;
gST->Hdr.CRC32 = 0;
Status = gBS->CalculateCrc32(
    &gST->Hdr,
    gST->Hdr.HeaderSize,
    &Crc32
);
gST->Hdr.CRC32 = OriginalCrc32;
if (EFI_ERROR (Status)) {
    return FALSE;
}
return (Crc32 == OriginalCrc32);
```

Example 73—Verify 32-bit CRC in UEFI System Table

5.2.9 ConvertPointer()

UEFI Boot Service drivers must never use this service.

This service may be required by UEFI *Runtime* Drivers if the UEFI Runtime Driver is required to convert pointer values that use physical addresses to pointer values that use virtual addresses. A UEFI Runtime driver must only call `ConvertPointer()` from an event notification function for an event of type `EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE` or a GUIDed event of type `EFI_EVENT_GROUP_VIRTUAL_ADDRESS_CHANGE`.

Caution: *Notification functions for events signaled when SetVirtualAddressMap() is called by an OS Loader or OS Kernel are not allowed to use any of the UEFI boot services, UEFI Console Services, or UEFI Protocol Services either directly or indirectly because those services are no longer available when SetVirtualAddressMap() is called. Instead, this type of notification function typically uses ConvertPointer() to convert pointers within data structures that are managed by the UEFI runtime driver from physical addresses to virtual addresses.*

UEFI system firmware takes care of most of the physical to virtual address translations that a UEFI Runtime Driver requires. For example, all of the code and data sections in the UEFI Runtime Driver image are automatically fixed up for proper execution at the virtual address ranges provided by the operating system when the operating system calls the UEFI Runtime Service `SetVirtualAddressMap()`.

If a UEFI Runtime Driver caches pointer values in global variables, or a UEFI Runtime Driver allocates buffers from `EfiRuntimeServicesData`, those pointer values must be converted from physical addresses to virtual address using the virtual address ranges provided by the operating system when the operating system calls the UEFI Runtime Service `SetVirtualAddressMap()`. If allocated buffers contain more pointers, then those pointer values must also be converted.

In these more complex scenarios, the order of the conversions is critical because the algorithm in the UEFI Runtime Driver must guarantee that no virtual addresses in the execution of the notification actually function because the event notification function on `SetVirtualAddressMap()` only executes in physical mode.

The following code fragment shows how a UEFI Runtime Driver can create an event whose notification function is executed in physical mode when the OS Loader or OS Kernel calls `SetVirtualAddressMap()`. There are two methods to create a `SetVirtualAddressMap()` event. This example shows the preferred method that uses `CreateEventEx()` to pass in the GUID of `gEfiEventVirtualAddressChangeGuid`. The alternate method uses `CreateEvent()` or `CreateEventEx()` with an event type of `EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE`. The created event is declared as a global variable, and makes the event available if the UEFI Runtime Driver needs to close the event if UEFI Runtime Driver is unloaded. The code fragments that follow this example show how `ConvertPointer()` may be used from `NotifySetVirtualAddressMap()`, the event notification function from this example.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

#include <Guid/EventGroup.h>

//
```

```

// Global variable for the SetVirtualAddressMap event
//
EFI_EVENT    mSetVirtualAddressMapEvent = NULL;

EFI_STATUS    Status;

//
// Create a Set Virtual Address Map event.
//
Status = gBS->CreateEventEx (
    EVT_NOTIFY_SIGNAL,                                // Type
    TPL_NOTIFY,                                      // NotifyTpl
    NotifySetVirtualAddressMap,                      // NotifyFunction
    NULL,                                            // NotifyContext
    &gEfiEventVirtualAddressChangeGuid, // EventGroup
    &mSetVirtualAddressMapEvent                         // Event
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 74—Create a Set Virtual Address Map event

The following code fragment shows how `ConvertPointer()` is used to convert a global variable functioning as a pointer from a physical address to that with a virtual address.

The flag `EFI_OPTIONAL_PTR` tells `ConvertPointer()` to not perform a conversion if the physical address of the pointer is `NULL`. This is useful if it is legal for some of the pointer values to be `NULL` and the `NULL` value needs to be preserved after the conversion. The only other legal value for this field is 0. The conversion should be performed unconditionally.

```

#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>

VOID *gGlobalPointer;

VOID
EFIAPI
NotifySetVirtualAddressMap (
    IN EFI_EVENT   Event,
    IN VOID        *Context
)
{
    EFI_STATUS    Status;

    Status = gRT->ConvertPointer (
        EFI_OPTIONAL_PTR,
        (VOID **) &gGlobalPointer
    );
}

```

Example 75—Convert a global pointer from physical to virtual

The code fragment in Example 76, below, is identical to 75, above, but uses the function `EfiConvertPointer()` from the EDK II library `UefiRuntimeLib` to call the UEFI Runtime Service `ConvertPointer()`.

```
#include <Uefi.h>
#include <Library/UefiRuntimeLib.h>

VOID *gGlobalPointer;

VOID
EFIAPI
NotifySetVirtualAddressMap (
    IN EFI_EVENT Event,
    IN VOID       *Context
)
{
    EFI_STATUS Status;

    Status = EfiConvertPointer (
        EFI_OPTIONAL_PTR,
        (VOID **) &gGlobalPointer
    );
}
```

Example 76—Using UefiRuntimeLib to convert a pointer

The EDK II library `UefiRuntimeLib` also provides the function `EfiConvertFunctionPointer()` to convert a function pointer from a physical address to a virtual address. On supported CPU architectures where there is no distinction between a data pointer and a function pointer, `EfiConvertPointer()` and `EfiConvertFunctionPointer()` are identical. On other CPU architectures such as IPF, where function calls are made through a `PLABEL`, converting a function pointer is more complex. The EDK II library `UefiRuntimeLib` helps hide these CPU specific details so the UEFI Driver sources can be the same for all supported CPU architectures.

Since the UEFI system firmware automatically converts functions in code sections of a UEFI Runtime Driver image from physical addresses to virtual addresses, `EfiConvertFunctionPointer()` is required only if a UEFI Driver caches a function pointer in a global variable or an allocated buffer.

```
#include <Uefi.h>
#include <Library/UefiRuntimeLib.h>

typedef
VOID
(EFIAPI * EFI_EXAMPLE_FUNCTION)(
    IN VOID   *Context
);

EFI_EXAMPLE_FUNCTION gGlobalFunctionPointer;

VOID
EFIAPI
NotifySetVirtualAddressMap (
    IN EFI_EVENT Event,
```

```

    IN VOID      *Context
)
{
EFI_STATUS  Status;

Status = EfiConvertFunctionPointer (
    EFI OPTIONAL_PTR,
    (VOID **) &gGlobalFunctionPointer
);
}

```

Example 77—Using UefiRuntimeLib to convert a function pointer

The EDK II library `UefiRuntimeLib` also provides helper function call `EfiConvertList()` to convert all the pointer values in a doubly linked list of type `LIST_ENTRY`. All the nodes in the linked list are traversed and the forward and backward link in each node is converted from a physical address to a virtual address.

Once this conversion is performed, the linked list cannot be accessed again in this function because all the pointer values are now virtual addresses. If the contents of the linked list contain structures with more pointer values that also need to be converted, those conversions must be performed prior to calling `EfiConvertList()`.

```

#include <Uefi.h>
#include <Library/UefiRuntimeLib.h>

LIST_ENTRY gGlobalList = INITIALIZE_LIST_HEAD_VARIABLE (gGlobalList);

VOID
EFI API
NotifySetVirtualAddressMap (
    IN EFI_EVENT Event,
    IN VOID      *Context
)
{
EFI_STATUS  Status;

Status = EfiConvertList (EFI OPTIONAL_PTR, &gGlobalList);
}

```

Example 78—Using UefiRuntimeLib to convert a linked list

5.2.10 InstallConfigurationTable()

This service is used to add, update, or remove an entry in the list of configuration table entries maintained in the UEFI System Table. These entries are typically used to pass information from the UEFI pre-boot environment to the operating system environment.

The configuration table entries are composed of a GUID and a pointer to a buffer. The GUID defines the type of memory that the buffer must use. If an operating system requires a configuration table entry that is allocated from a memory type that is not preserved after `ExitBootServices()`, then the OS Loader or OS Kernel must make a copy of the data structure prior calling `ExitBootServices()`.

A UEFI Driver has a limited set of options to pass information into the operating system environment. These include:

- Protocols
- UEFI Variables
- Configuration Table Entries

The services required to locate protocols in the Handle Database are not available after `ExitBootServices()`, so information passed up through protocols must be located by the OS Loader or OS Kernel prior to calling `ExitBootServices()`. UEFI Variables are good for small amounts of data, but may consume the scarce variable resources and access to variable storage may be slower than system memory. A configuration table entry is good for larger amounts of data generated each boot and it is stored in system memory. The *UEFI Specification* defines a set of GUIDs for standard configuration table entries that includes:

- ACPI Tables
- SMBIOS Tables
- SAL System Table (IPF only)
- MPS Tables
- Debug Image Info Tables
- Image Execution Information Table
- Exported HII Database
- User Information Table
- Capsules
- UNDI Configuration Table

Most of these usages are handled by the UEFI system firmware. The one usage impacting UEFI Drivers is the UNDI Configuration Table that is produced by a UEFI UNDI Driver for a Network Interface Controller (NIC). UEFI Drivers are allowed to define new GUIDs for new configuration table entries to pass information from the UEFI pre-boot environment to the OS environment.

The following code fragment shows how an UNDI driver can add or update an UNDI Configuration Table entry to the list of configuration table entries maintained in the UEFI System Table.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>

EFI_STATUS      Status;
UNDI_CONFIG_TABLE *UndiConfigTable;

// 
// Allocate and zero UNDI_CONFIG_TABLE from EfiRuntimeServicesData
```

```

// UndiConfigTable = (UNDI_CONFIG_TABLE *)AllocateRuntimeZeroPool (
//                     sizeof (UNDI_CONFIG_TABLE)
// );
//
// Initialize UNDI_CONFIG_TABLE
//
//
// Add or update a configuration table
//
Status = gBS->InstallConfigurationTable (
    &gEfiNetworkInterfaceIdentifierProtocolGuid_31,
    &UndiConfigTable
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 79—Add or update a configuration table entry

The code fragment below shows how an UNDI driver can remove an UNDI Configuration Table entry from the list of configuration table entries maintained in the UEFI System Table.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS Status;

//
// Remove a configuration table
//
Status = gBS->InstallConfigurationTable (
    &gEfiNetworkInterfaceIdentifierProtocolGuid_31,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 80—Add or update a configuration table entry

5.2.10.1 **WaitForEvent()**

This service stops execution until an event is signaled and is only allowed to be called at a priority level of **TPL_APPLICATION**. This means that **WaitForEvent()** may not be used from an event notification function because event notification functions always execute at priority levels above **TPL_APPLICATION**. If a UEFI Driver needs to know the current state of an event, the **CheckEvent()** service should be used instead of **WaitForEvent()**.

WaitForEvent() may be used by UEFI Applications. The typical use case is to wait for input from a device such as a keyboard or mouse as part of a user interface. There are a few older protocols that UEFI Drivers may produce that interact with the user and the

implementation of these protocols could use `WaitForEvent()`. For example, the `SetOptions()` function in the Driver Configuration Protocol.

The following code fragment shows how `WaitForEvent()` is used to wait for one of two events to be signaled. One event is signaled if a key is pressed on the console input device from the UEFI System Table. The other event is a one-shot timer that is signaled after waiting for 1 second. `WaitForEvent()` does not return until either a key is pressed or 1 second has passed. This can be used to wait for a key and also update the console with status information once a second. Status is set to `EFI_SUCCESS` if a key is pressed and `EFI_TIMEOUT` if no key is pressed.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS  Status;
EFI_EVENT   WaitList[2];
UINTN      Index;

//
// Add ConIn event from the UEFI System Table to the array of events
//
WaitList[0] = &gST->ConIn->WaitForKey;

//
// Add timer event that fires in 1 second to the array of events
//
Status = gBS->CreateEvent (
    EVT_TIMER | EVT_NOTIFY_WAIT,    // Type
    TPL_NOTIFY,                  // NotifyTpl
    NULL,                        // NotifyFunction
    NULL,                        // NotifyContext
    &WaitList[1]                 // Event
);
if (EFI_ERROR (Status)) {
    return Status;
}

Status = gBS->SetTimer (
    WaitList[1],
    TimerRelative,
    EFI_TIMER_PERIOD_SECONDS (1)
);
if (EFI_ERROR (Status)) {
    gBS->CloseEvent (WaitList[1]);
    return Status;
}

//
// Wait for the console input or the timer to be signaled
//
Status = gBS->WaitForEvent (2, WaitList, &Index);

//
// Close the timer event
//
gBS->CloseEvent (WaitList[1]);
```

```

// If the timer event expired return EFI_TIMEOUT
//
if (!EFI_ERROR (Status) && Index == 1) {
    Status = EFI_TIMEOUT;
}

```

Example 81—Wait for key press or timer event

5.2.11 GetNextMonotonicCount()

This service provides a 64-bit monotonically increasing counter that is guaranteed to provide a higher value each time `GetNextMonotonicCount()` is called. This 64-bit counter is not related to any time source, so *this service should never be used for delays, polling, or for any type of time measurement.*

`GetNextHighMonotonicCount()` is related to this same 64-bit monotonic counter, but that service is only intended to be used by operating systems after `ExitBootServices()` is called to manage the non-volatile upper 32-bits of the 64-bit monotonic counter. A UEFI Driver should only use the UEFI Boot Service `GetNextMonotonicCount()` because it manages all 64-bits of the monotonic counter before `ExitBootServices()` is called.

The code fragment below show how `GetNextMonotonicCount()` can be used to retrieve the next 64-bit value for the monotonic counter.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS  Status;
UINT64      MonotonicCount;

Status = gBS->GetNextMonotonicCount (&MonotonicCount);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 82—Retrieve 64-bit monotonic counter value.

5.3 Services that UEFI drivers should not use

The following table lists the UEFI services that should *not* be used by UEFI drivers. These services may be used by components other than UEFI Drivers, or these services may have been replaced by newer services and should no longer be used by UEFI Drivers. The following sections describe why each of these services should not be used in UEFI drivers and are grouped by Service Type.

Table 20—UEFI services that should not be used by UEFI drivers

Service	Type	Service Type	Notes
InstallProtocolInterface()	Boot	Protocol Handler	Installs a protocol interface on a device handle. Replaced by InstallMultipleProtocolInterfaces().
UninstallProtocolInterface()	Boot	Protocol Handler	Removes a protocol interface from a device handle. Replaced by UninstallMultipleProtocolInterfaces().
HandleProtocol()	Boot	Protocol Handler	Queries a handle to determine if it supports a specified protocol. Replaced by OpenProtocol().
LocateHandle()	Boot	Protocol Handler	Returns an array of handles that support a specified protocol. This service has been replaced by LocateHandleBuffer().
ProtocolsPerHandle()	Boot	Protocol Handler	Retrieves the list of protocols installed on a handle. The return buffer is automatically allocated. This service has been replaced with: The <i>Start</i> function in the UEFI Driver Binding Protocol.
RegisterProtocolNotify()	Boot	Protocol Handler	Registers an event that is to be signaled whenever an interface is installed for a specified protocol. This service has been replaced with: The <i>Supported()</i> function in the UEFI Driver Binding Protocol.
UnloadImage()	Boot	Image	Used to unload a previously loaded UEFI Driver.
GetNextVariableName()	Runtime	Variable	Used to walk the list of UEFI variables that are maintained through the UEFI variable services. Use of this service is not usually necessary.
SetWatchDogTimer()	Boot	Time-related	Sets the current local time and date information. UEFI drivers should not use this service; UEFI drivers should not modify the system time or the wakeup timer.
SetTime()	Runtime	Time-related	Sets the current local time and date information. UEFI drivers should not use this service; UEFI drivers should not modify the system time or the wakeup timer.
GetWakeupTime()	Runtime	Time-related	Returns the current wakeup alarm clock setting. UEFI drivers should not use this service; the watchdog timer is managed from the UEFI boot manager.

Service	Type	Service Type	Notes
SetWakeupTime()	Runtime	Time-related	Sets the system wakeup alarm clock time. UEFI drivers should not use this service; the watchdog timer is managed from the UEFI boot manager.
GetMemoryMap()	Boot	Memory Allocation	Returns the current boot services memory map and memory map key.
ExitBootServices()	Boot	Special	This service hands control of the platform from the UEFI conformant firmware to an OS. UEFI drivers must never use this service.
SetVirtualAddressMap()	Runtime	Special	This service is used only by UEFI OS loaders or OS kernels for operating systems that wish to call UEFI runtime services using virtual addresses. UEFI drivers must never use this service.
QueryCapsuleCapabilities()	Runtime	Special	Test to see if a capsule or capsules can be updated via UpdateCapsule().
UpdateCapsule()	Runtime	Special	Allows the operating system to pass information to firmware.
ResetSystem()	Runtime	Special	Resets and sets a watchdog timer used during boot services time. UEFI drivers should not use this service; the watchdog timer is managed from the UEFI boot manager.
Exit()	Boot	Special	UEFI drivers should not use this service. This service is typically used by applications.
GetNextHighMonotonicCount()	Runtime	Special	Provides a 64-bit monotonic counter that is guaranteed to increase.

5.3.1 **InstallProtocolInterface()**

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

5.3.2 **UninstallProtocolInterface()**

This service removes one protocol interface from a handle in the handle database. The functionality of this service has been replaced by **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.

5.3.3 HandleProtocol()

UEFI drivers should not use this service because a UEFI drivers that uses this service to lookup protocol is not conformant with the UEFI Driver Model. Instead, [OpenProtocol\(\)](#) should be used because it provides equivalent functionality, and it also allows the Handle Database to track the components that are using different protocol interfaces in the handle database.

5.3.4 LocateHandle()

This service returns an array of handles that support a specified protocol. This service requires the caller to allocate the return buffer. The [LocateHandleBuffer\(\)](#) service is easier to use and produces smaller executables because it allocates the return buffer for the caller.

5.3.5 ProtocolsPerHandle()

This service retrieves the list of protocols that are installed on a handle. In general, UEFI drivers know what protocols are installed on the handles that the UEFI driver is managing, so this service is not required for proper UEFI Driver operation. This service is typically used by UEFI applications, such as diagnostics or debug utilities, that need to traverse the entire contents of the Handle Database.

5.3.6 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. The *EFI 1.10 Specification* introduced the UEFI Driver Model, and is still supported in the current versions of the *UEFI Specification*. The UEFI Driver Model provides a more flexible mechanism for a driver to layer on top of another driver that eliminated the need for [RegisterProtocolNotify\(\)](#). The [RegisterProtocolNotify\(\)](#) service is still supported for compatibility with previous versions of the *EFI Specification*.

5.3.7 UnloadImage()

This service unloads a UEFI Driver from memory that was previously loaded using the UEFI Boot Service [LoadImage\(\)](#). There are currently no known use cases for this service from a UEFI Driver. [UnloadImage\(\)](#) is typically used from a UEFI Application like the UEFI Shell to manage the set of active UEFI Drivers.

Caution: A UEFI Driver must never use this service to unload itself. This service frees all the memory associated with the UEFI Driver and returns control to the location the UEFI Driver used to reside in memory, thereby producing unexpected results.

5.3.8 GetNextVariableName()

This service is used to traverse the list of UEFI variables that are maintained through the UEFI Variable Services. Since, in general, UEFI drivers know the specific UEFI variables that the UEFI Driver is required to access, there is no need for a UEFI driver to traverse the list of all the UEFI variables. This service is typically used by UEFI applications, such as a diagnostic or a debug utility, to show the contents of all the UEFI Variables present in a platform.

The example below shows how the `GetNextVariableName()` service can be used to traverse and print the entire contents of the UEFI variable store. It uses the EDK II `MemoryAllocationLib` to allocate, reallocate, and free buffers; the EDK II `UefiLib` to print formatted strings to the UEFI console output device; and the EDK II `UefiRuntimeServicesTableLib` to call the `GetNextVariableName()` and `GetVariable()` runtime services.

```
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Library/UefiRuntimeServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>

EFI_STATUS Status;
EFI_GUID Guid;
UINTN NameBufferSize;
UINTN NameSize;
CHAR16 *Name;
UINTN DataSize;
UINT8 *Data;
UINTN Index;

//
// Initialize the variable name and data buffer variables
// to retrieve the first variable name in the variable store
//
NameBufferSize = sizeof (CHAR16);
Name = AllocateZeroPool (NameBufferSize);

//
// Loop through all variables in the variable store
//
while (TRUE) {
    //
    // Loop until a large enough variable name buffer is allocated
    //
    do {
        NameSize = NameBufferSize;
        Status = GRT->GetNextVariableName (&NameSize, Name, &Guid);
        if (Status == EFI_BUFFER_TOO_SMALL) {
            //
            // Grow the buffer Name to NameSize bytes
            //
            Name = ReallocatePool (NameBufferSize, NameSize, Name);
            if (Name == NULL) {
                return EFI_OUT_OF_RESOURCES;
            }
            NameBufferSize = NameSize;
        }
    }
}
```

```

} while (Status == EFI_BUFFER_TOO_SMALL);

//
// Exit main loop after last variable name is retrieved
//
if (EFI_ERROR (Status)) {
    FreePool (Name);
    return Status;
}

//
// Print variable guid and name
//
Print (L"%g : %s", &Guid, Name);

//
// Initialize variable data buffer as an empty buffer
//
DataSize = 0;
Data     = NULL;

//
// Loop until a large enough variable data buffer is allocated
//
do {
    Status = gRT->GetVariable (Name, &Guid, NULL, &DataSize, Data);
    if (Status == EFI_BUFFER_TOO_SMALL) {
        //
        // Allocate new buffer for the variable data
        //
        Data = AllocatePool (DataSize);
        if (Data == NULL) {
            FreePool (Name);
            return EFI_OUT_OF_RESOURCES;
        }
    }
} while (Status == EFI_BUFFER_TOO_SMALL);

if (EFI_ERROR (Status)) {
    FreePool (Data);
    FreePool (Name);
    return Status;
}

//
// Print variable data
//
for (Index = 0; Index < DataSize; Index++) {
    if ((Index & 0x0f) == 0) {
        Print (L"\n  ");
    }
    Print (L"%02x ", Data[Index]);
}
Print (L"\n");

FreePool (Data);
}

```

Example 83—Print all UEFI variable store contents

5.3.9 SetWatchdogTimer()

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

5.3.10 SetTime(), GetWakeupTime(), and 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.

5.3.11 GetMemoryMap()

UEFI drivers should not use this service because UEFI drivers should not depend upon the physical memory map of the platform. The `AllocatePool()` and `AllocatePages()` services allow a UEFI driver to allocate system memory. The `FreePool()` and `FreePages()` services 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 of 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.

This service is typically used by a UEFI OS Loader to retrieve the memory map just before the OS takes control of the platform by calling `ExitBootServices()`. It may also be used by UEFI applications, such as diagnostics or debug utilities, to show how platform memory has been allocated.

5.3.12 ExitBootServices()

This service hands control of the platform from UEFI conformant firmware to a UEFI conformant operating system. It should be invoked only by UEFI OS loaders or OS kernels. It should never be called by a UEFI driver. Refer to the Image Services section in the *UEFI Specification* for more information about this service.

5.3.13 SetVirtualAddressMap()

This service is used only by UEFI OS loaders or OS kernels when an operating system requests UEFI Runtime Services be mapped using virtual addresses. It must be called after `ExitBootServices()` is called. As a result, it is not legal for EFI drivers to call this service.

5.3.14 QueryCapsuleCapabilities()

UEFI drivers should not use this service. It is typically used by an operating system or an OEM provided utility to test to see if a capsule or capsules can be updated via `UpdateCapsule()` service as part of a capsule update action.

5.3.15 UpdateCapsule()

UEFI drivers should not use this service. It is typically used by an operating system or an OEM provided utility to pass a capsule to the firmware as part of a capsule update action.

5.3.16 ResetSystem()

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

The following code fragment shows how the UEFI Runtime Service `ResetSystem()` is used to request a warm reset of the platform. The EDK II library `UefiRuntimeServicesTableLib` provides a global variable for the UEFI Runtime Services Table for the currently executing driver. In this example, the global variable for the UEFI Runtime Services Table, `gRT`, is used to call the UEFI Runtime Service `ResetSystem()`.

```
#include <Uefi.h>
#include <Library/UefiRuntimeServicesTableLib.h>
#include <Library/BaseLib.h>

//
// Perform a warm reset of the platform
//
gRT->ResetSystem (EfiResetWarm, EFI_SUCCESS, 0, NULL);

//
// Halt.  ResetSystem should never return.
//
CpuDeadLoop ();
```

Example 84—ResetSystem

5.3.17 Exit()

The `Exit()` service is typically only used by UEFI applications. UEFI drivers usually have simple driver entry point implementations and typically return an `EFI_STATUS` code from their entry point function. This is the recommended style for UEFI driver implementations. If `EFI_SUCCESS` is returned by a UEFI driver, then the UEFI driver remains loaded in system memory. If an error status is returned, then the UEFI driver is unloaded from system memory.

The `Exit()` service allows a UEFI image to exit without having to return an `EFI_STATUS` value from the UEFI image's entry point. A UEFI driver with more complex logic in its entry point may discover a condition that requires the UEFI driver to exit immediately. In this rare condition, the `Exit()` service could be used. However, the UEFI driver implementation must take care to free any allocated resources and uninstall all protocols before returning an error code through the `Exit()` service. The following example shows how the `Exit()` service could be used by a UEFI driver to exit with a status code of `EFI_UNSUPPORTED`. The EDK II library `UefiBootServicesTableLib` provides the global `gBS`—a pointer to the UEFI Boot Services Table and `gImageHandle`—the Image Handle of the currently executing UEFI image.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>

//
// Exit the current UEFI image with a status of EFI_UNSUPPORTED
//
gBS->Exit (gImageHandle, EFI_UNSUPPORTED, 0, NULL);
```

Example 85—Exit from a UEFI Driver

5.3.18 GetNextHighMonotonicCount()

There are no use cases for this service by a UEFI Driver. It should never be called.

This service is only used by operating systems to manage the upper 32-bits of the 64-bit monotonic counter after the operating system has called `ExitBootServices()`. An operating system that chooses to use the UEFI provided 64-bit monotonic counter should acquire the value of the 64-bit monotonic counter before `ExitBootServices()` using the UEFI Boot Service `GetNextMonotonicCount()`. The operating system can manage the volatile lower 32-bits of the 64-bit monotonic counter on its own. If a 32-bit rollover condition occurs, then the operating system can use the UEFI Runtime Service `GetNextHighMonotonicCount()` to increment the upper 32-bits of the 64-bit monotonic counter. The upper 32-bits are non-volatile and it is the responsibility of the UEFI firmware to guarantee that the upper 32-bits of the 64-bit monotonic counter are preserved across system resets and power cycles.

6

UEFI Driver Categories

The different categories of UEFI drivers are introduced in [Chapter 3](#) of this guide. These driver categories are discussed throughout this document, but emphasis is placed on drivers that follow the UEFI driver model because they are the most commonly implemented. The driver categories that follow the UEFI driver model include:

- Device drivers
- Bus drivers
- Hybrid drivers

There are several subtypes and optional features for the three categories of drivers. This chapter introduces the subtypes and optional features of drivers that follow the UEFI driver model. Understanding the different categories of UEFI drivers helps driver writers identify the category of driver to implement and the algorithms used in their implementation. The less common service drivers, root bridge drivers and initializing drivers are also discussed. [Appendix B](#) contains a table of example drivers from the EDK II along with the features that each implement.

6.1 Device drivers

All device drivers following the UEFI driver model share a set of common characteristics. The next two sections describe the required and optional features for device drivers. These sections are followed by a detailed description of device drivers that produce both single and multiple instances of the Driver Binding Protocol

6.1.1 Required Device Driver Features

Device drivers are required to implement the following features:

- A driver entry point that installs one or more instances of the Driver Binding Protocol.
- Manages one or more controller handles. Even if a driver writer is convinced that the driver manages only a single controller, it is strongly recommended that the driver be designed to manage multiple controllers. The overhead for this functionality is low, and it makes 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 the 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 Optional Device Driver Features

The following lists features that a device driver can optionally implement.

- Install one or more instances of the `EFI_COMPONENT_NAME2_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 manages.

- Register one or more HII packages in the driver's entry point.

HII packages provide strings, fonts, and forms that allow users (such as IT administrators) to change the driver's configuration. They are only required if a driver must provide the ability for a user to change configuration settings for a device.

- Install one or more instances of the `EFI_DRIVER_DIAGNOSTICS2_PROTOCOL` in the driver's entry point.

If a driver needs to provide diagnostics for the controllers that the driver manages, this protocol is required.

- Provide 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.

- Install one or more instances of the `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL` in the driver's entry point.

This protocol is required only if a driver needs a higher priority rule for connecting drivers to controllers through the UEFI Boot Service `ConnectController()`.

- Install one or more instances of the `EFI_DRIVER_HEALTH_PROTOCOL` in the driver's entry point.

This protocol is only required for drivers that manage devices that can be in a bad state that is recoverable through either a repair operation or a configuration operation.

- Install an instance of the `EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL` in the driver's entry point.

This protocol is required for PCI controller or other plug-in cards. Implementation of this feature is **recommended**.

- Create 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 a UEFI runtime driver.

6.1.3 Compatibility with Older EFI/UEFI Specifications

The following lists features that a device driver can optionally implement to provide compatibility with older versions of the *EFI* and *UEFI Specifications*.

- Install one or more instances of the `EFI_COMPONENT_NAME_PROTOCOL` in the driver's entry point.

Implementing this feature is **strongly recommended** for drivers required to be compatible with EFI 1.10. It allows a driver to provide human-readable names for the name of the driver and the controllers that the driver is managing. The EDK II libraries provide easy methods to produce both the Component Name Protocol and the Component Name 2 Protocol with very little additional overhead.

- Installs one or more instances of the `EFI_DRIVER_CONFIGURATION_PROTOCOL` in the driver's entry point.

If a driver must be compatible with EFI 1.10, and has any configurable options, this protocol is required.

- Installs one or more instances of the `EFI_DRIVER_CONFIGURATION2_PROTOCOL` in the driver's entry point.

If a driver must be compatible with UEFI 2.0 and has any configurable options, this protocol is required.

- Install one or more instances of the `EFI_DRIVER_DIAGNOSTICS_PROTOCOL` in the driver's entry point.

If a driver must be compatible with EFI 1.10 and provide diagnostics for the controllers that the driver manages, this protocols is required.

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 among those 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 `LoadImage()` Boot Service and invoked with the `StartImage()` Boot Service. The `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 from where the device driver was loaded and the location in system memory to where the device driver was placed.

The `Unload()` service of the `EFI_LOADED_IMAGE_PROTOCOL` is initialized to `NULL` by `LoadImage()`. This setting means that by default the driver does not have an unload function.

The `StartImage()` service transfers control to the driver's entry point as described in the PE/COFF header of the UEFI Driver image. The PE/COFF header layout is defined in the *Microsoft Portable Executable and Common Object File Format Specification*.

The driver entry point is responsible for installing the Driver Binding Protocol onto the driver's image handle. The figure 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.

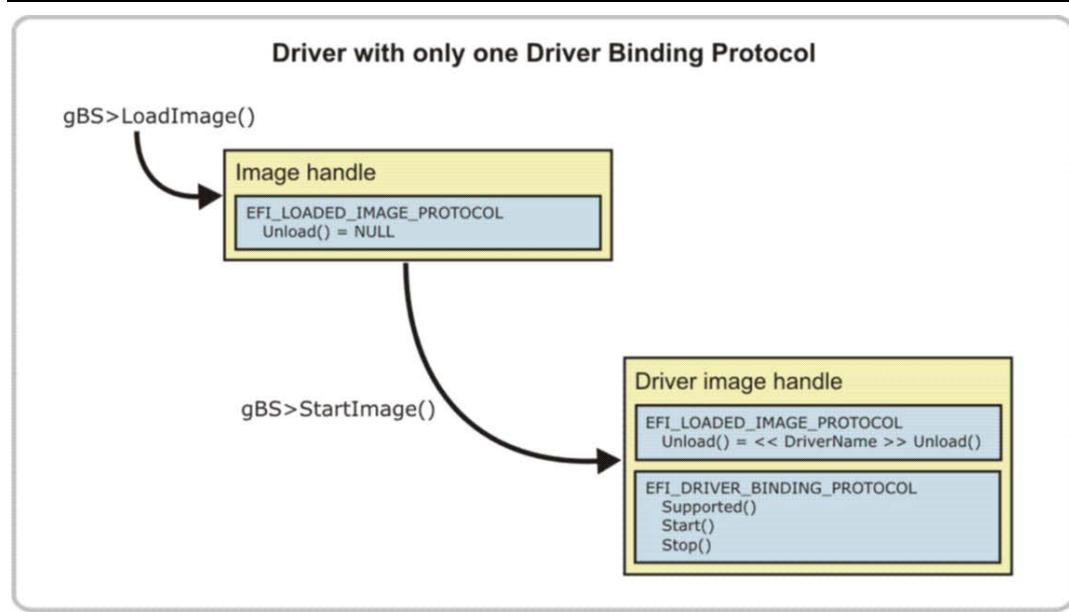


Figure 9—Device driver with single Driver Binding Protocol

The [following figure](#) is the same as the figure above, except this device driver has also implemented 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. These are part of the driver's initialization (the driver's entry point).

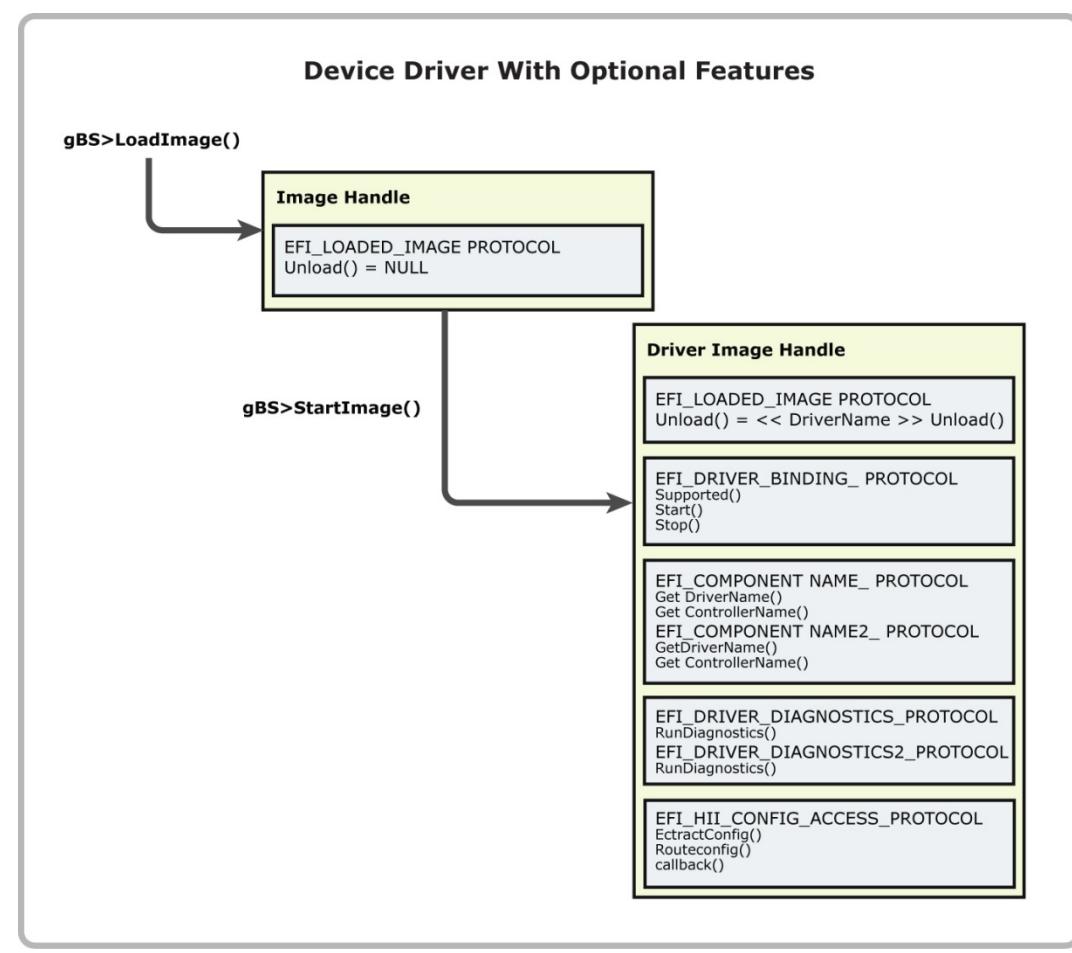


Figure 10—Device driver with optional features

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 driver binding protocol. The first instance of **EFI_DRIVER_BINDING_PROTOCOL** is installed onto the driver's image handle, and the additional instances of the Driver Binding Protocol are installed onto newly created driver binding handles.

The [figure 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 Driver Binding Protocol.

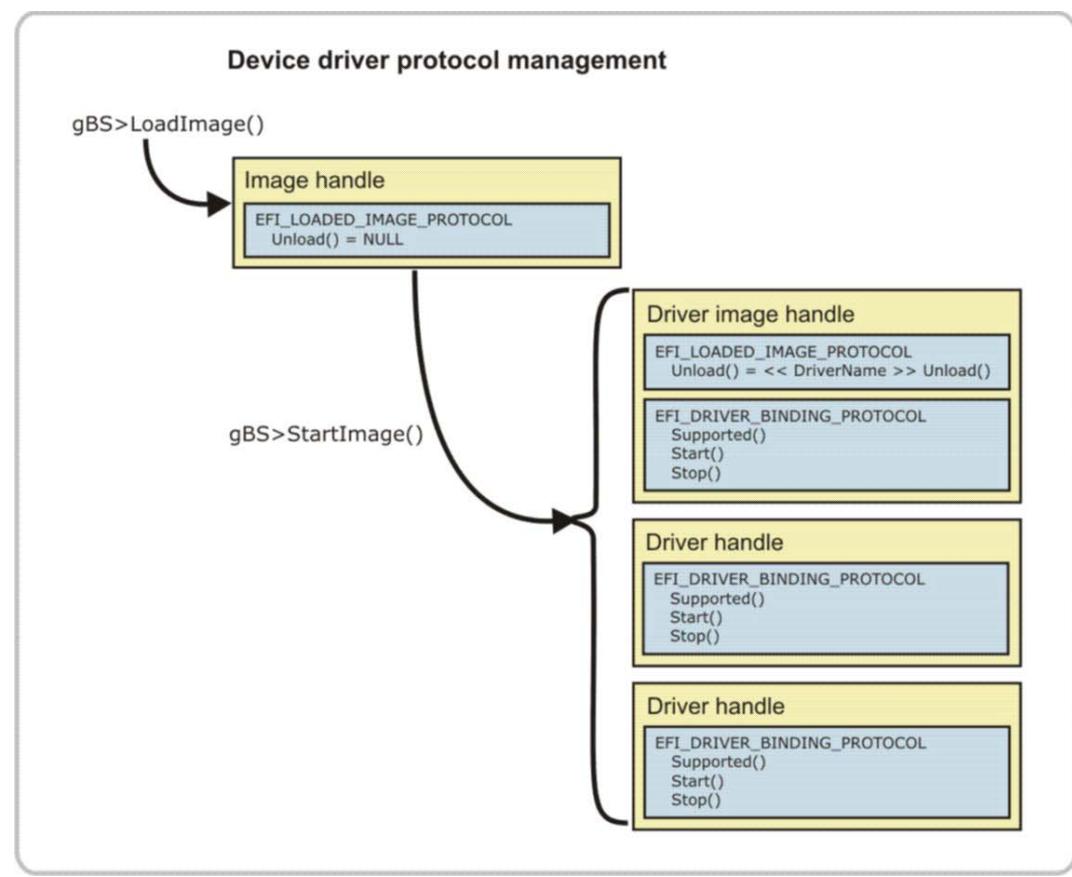


Figure 11—Device driver with multiple Driver Binding Protocols

Any device driver that produces multiple instances of the **EFI_DRIVER_BINDING_PROTOCOL** can be broken up into multiple drivers. Each driver would then produce a single instance of the **EFI_DRIVER_BINDING_PROTOCOL**. However, there are advantages if a driver produces multiple instances of the Driver Binding Protocol.

First, 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.

Combining drivers can also help manage platform features. A single platform's features may require several drivers. If the drivers are separate, multiple drivers have to be dealt with individually to add or remove that single feature.

An example device driver in EDK II that produces multiple instances of the Driver Binding Protocol is the console platform driver in the **MdeModulePkg/Universal/Console/ConPlatformDxe** subdirectory. This driver implements the platform policy for managing multiple console input and output devices. It produces one Driver Binding Protocol for the console output devices, and another 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 Driver Binding Protocol are responsible for opening the I/O-related protocols being consumed using the EFI Boot Service `OpenProtocol()`. The `Stop()` function is responsible for closing the consumed I/O-related protocols using `CloseProtocol()`.

A protocol can be opened in several different modes, but the most common is `EFI_OPEN_PROTOCOL_BY_DRIVER`. When a protocol is opened by `EFI_OPEN_PROTOCOL_BY_DRIVER`, a test is made to see if that protocol is already being consumed by any other drivers. The open operation succeeds only if the protocol is not being consumed by any other drivers.

Caution: Using the `OpenProtocol()` service with `EFI_OPEN_PROTOCOL_BY_DRIVER` is how resource conflicts are avoided in the UEFI driver model. However, it requires that every driver present in the system follow the driver interoperability rules for all resource conflicts to be avoided.

The [following figure](#) shows the image handle for a device driver as `LoadImage()` and `StartImage()` are called. In addition, it shows the states of three different controller handles as the 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 `EFI_OPEN_PROTOCOL_BY_DRIVER`, and **Protocol B** is installed onto the same controller handle. The implementation of **Protocol B** uses 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 their previous state, before `start()` was called, so the `stop()` service uninstalls **Protocol B** and closes **Protocol A**.

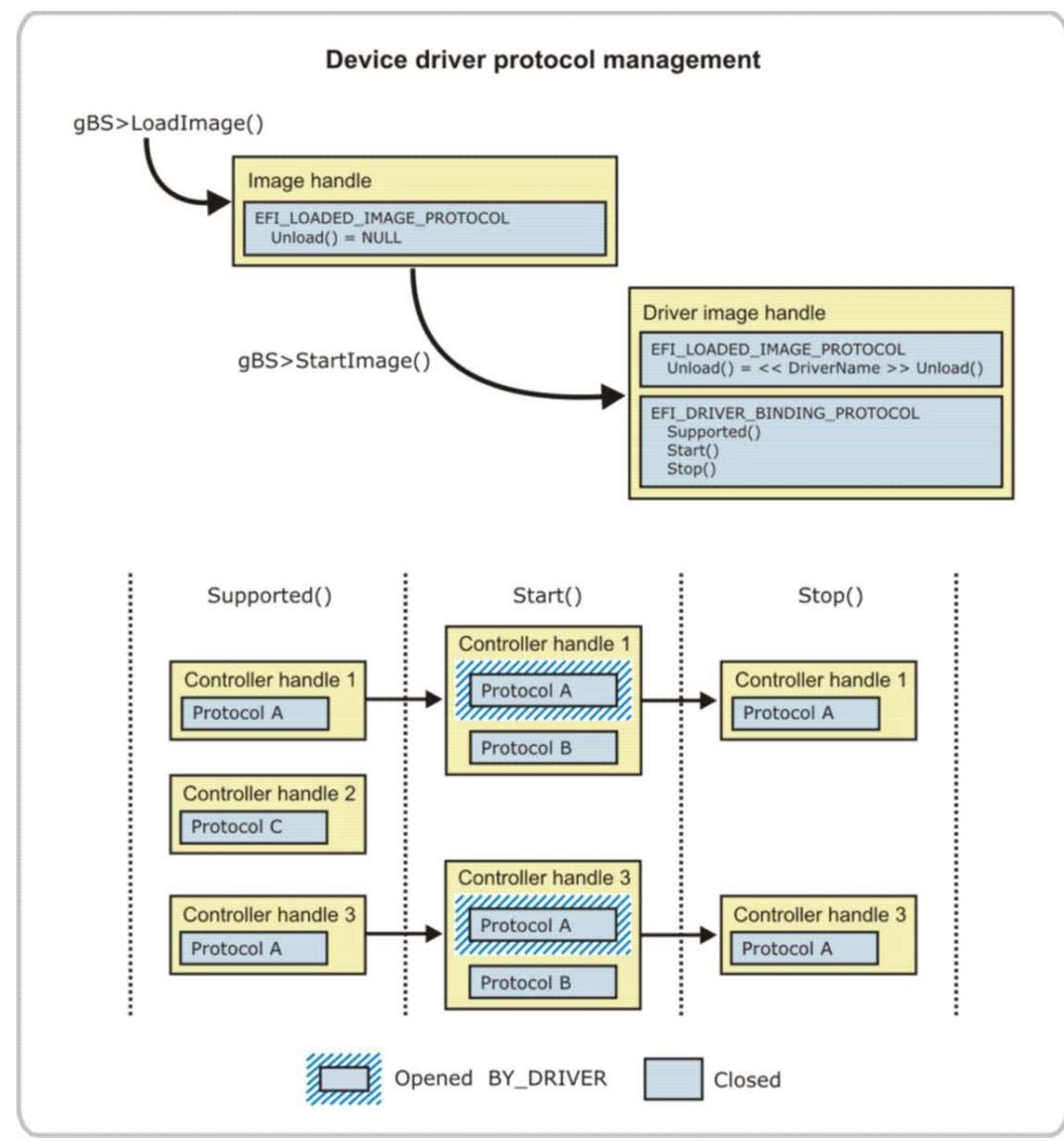


Figure 12—Device driver protocol management

The [figure below](#) shows a more complex device driver that requires **Protocol A** and **Protocol B** to produce **Protocol C**. Notice that the controller handles that do not support *either* **Protocol A** or **Protocol B** do not pass the `Supported()` test. In addition, controller handles that *only* support **Protocol A** or **Protocol B** also do not pass the `Supported()` test. Finally, note that **Controller Handle 6** already has **Protocol A** opened by `EFI_OPEN_PROTOCOL_BY_DRIVER`, so this device driver requiring both **Protocol A** and **Protocol B** also does not pass the `Supported()` test.

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 driver support for a specific controller handle.

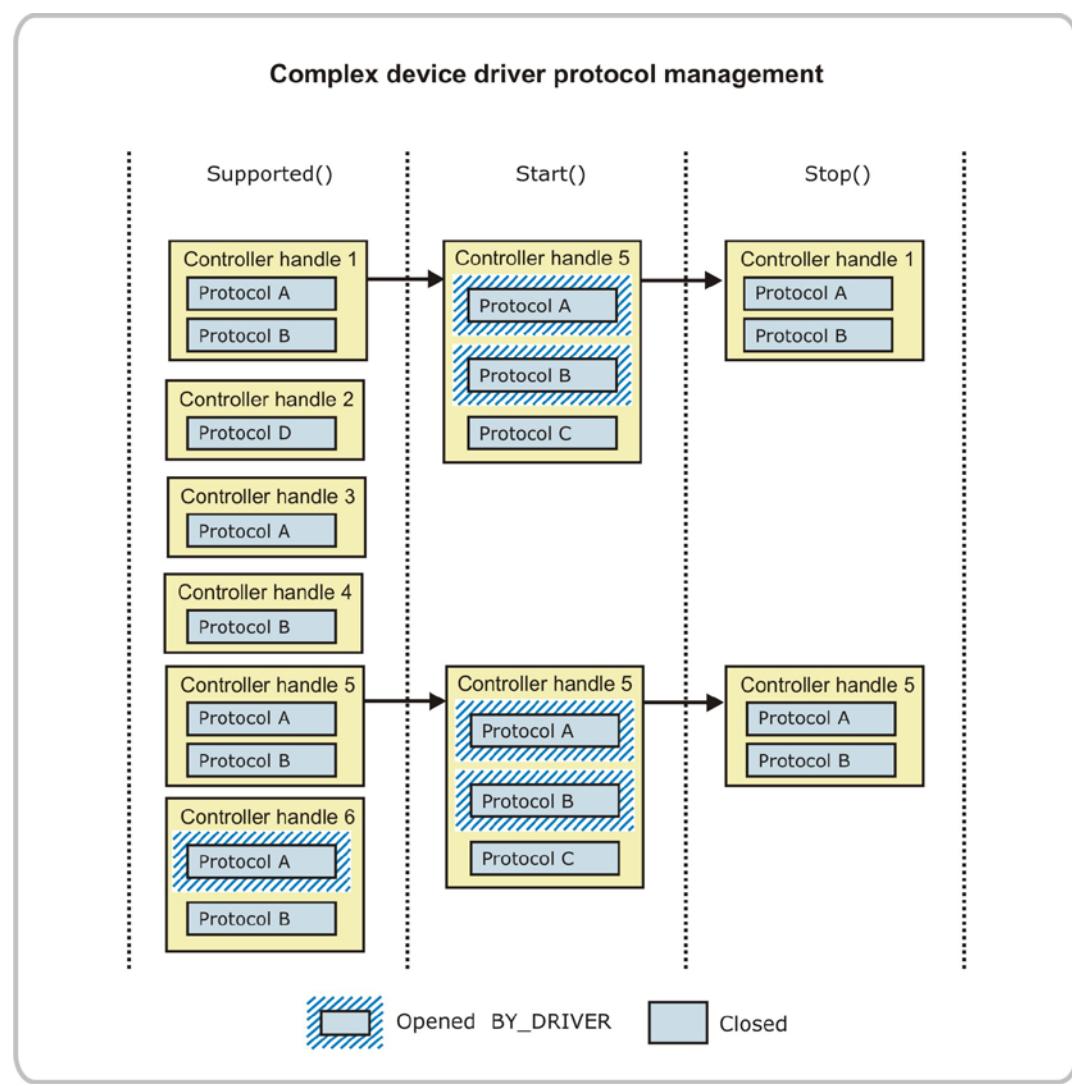


Figure 13—Complex device driver protocol management

TIP: 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 Driver Binding Protocol.

This scenario is another way that a device driver may be required to produce multiple instances of the Driver Binding Protocol. The `Supported()` service for each Driver Binding Protocol attempts 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 discussions describe the required and optional features for bus drivers. These sections are followed by a detailed description of bus drivers that do the following:

- Produce a single instance of the Driver Binding Protocol
- Produce multiple instances of the Driver Binding Protocol
- Produce all of their child devices in their **start()** function
- Are able to produce a single child device in their **start()** function
- Produce at most one child device from their **start()** function
- Bus drivers that do not produce any child devices in their **start()** function
- Produce child devices with multiple parent devices

6.2.1 Required Bus Driver Features

Bus drivers are required to implement the following features:

- Install one or more instances of the **EFI_DRIVER_BINDING_PROTOCOL** in the driver's entry point.
- Manage one or more controller handles. Even if a driver writer is convinced that the driver manages only a single bus controller, the driver should be designed to manage multiple bus controllers. The overhead for this functionality is low, and it makes the driver more portable.
- Produce any child handles. This feature is the key distinction between device drivers and bus drivers. (Device drivers do not produce child handles.)
- 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

The following lists features that a bus driver can optionally implement. Optional recommended features are noted below.

- Install one or more instances of the `EFI_COMPONENT_NAME2_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.
- Register one or more HII packages in the driver's entry point. HII packages provide strings, fonts, and forms that allow users (such as IT administrators) to change the driver's configuration. HII packages are only required if a driver must provide the ability for a user to change configuration settings for a device.
- Install one or more instances of the `EFI_DRIVER_DIAGNOSTICS2_PROTOCOL` in the driver's entry point. If a driver needs to provide diagnostics for the controllers that it manages, this protocol is required.
- Provide 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.
- Parses the `RemainingDevicePath` parameter that is passed into the `Supported()` and `Start()` services of the Driver Binding Protocol if it is not `NULL`. This is **strongly recommended** so a bus driver can start only the one child specified by `RemainingDevicePath`. Implementing this feature may significantly improve platform boot performance.
- Install an `EFI_DEVICE_PATH_PROTOCOL` on each child handle that is created. This 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.
- Install one or more instances of the `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL` in the driver's entry point. This protocol is required only if a higher priority rule for connecting drivers to a controller through the UEFI Boot Service `ConnectController()` is needed.
- Install one or more instances of the `EFI_DRIVER_HEALTH_PROTOCOL` in the driver's entry point. This protocol is required only for drivers that manage devices that can be in a bad state that is recoverable through either a repair operation or configuration operation.
- Install one or more instances of the `EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL` in the driver's entry point. This protocol is required only with bus drivers for a bus type where the devices on the bus can provide a container for more than one UEFI Driver. An example bus type is PCI where PCI Option ROMs on PCI Adapters may contain more than one UEFI Driver.

- Install an instance of the `EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL` in the driver's entry point. This protocol is required for PCI controllers or other plug-in cards. Implementation of this feature is **recommended**.
- Create 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 a UEFI runtime driver.

6.2.2.1

Compatibility with Older EFI/UEFI Specifications

The following lists the features a bus driver can optionally implement to provide compatibility with older versions of the *EFI Specification* and *UEFI Specification*.

- Install one or more instances of the `EFI_COMPONENT_NAME_PROTOCOL` in the driver's entry point. Implementing this feature is **strongly recommended** for drivers that are required to be compatible with EFI 1.10. It allows a driver to provide human-readable names for the name of the driver and the controllers the driver manages. The EDK II libraries provide easy methods to produce both the Component Name Protocol and the Component Name 2 Protocol with very little additional overhead.
- Install one or more instances of the `EFI_DRIVER_CONFIGURATION_PROTOCOL` in the driver's entry point. If a driver must be compatible with EFI 1.10, and has any configurable options, this protocol is required.
- Install one or more instances of the `EFI_DRIVER_CONFIGURATION2_PROTOCOL` in the driver's entry point. If a driver must be compatible with UEFI 2.0 and has any configurable options, this protocol is required.
- Install one or more instances of the `EFI_DRIVER_DIAGNOSTICS_PROTOCOL` in the driver's entry point. If a driver must be compatible with EFI 1.10 and provide diagnostics for the controllers that the driver manages, this protocol is required.

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 Driver Binding Protocol. The following sections describe the behaviors of the `Start()` function of the Driver Binding Protocol for each type of bus driver.

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 Driver Binding Protocol. The following discussions describe the behaviors of the `Start()` function of the Driver Binding Protocol for each type of bus driver.

An example bus driver in EDK II that produces multiple instances of the `EFI_DRIVER_BINDING_PROTOCOL`, is the console splitter driver in the `MdeModulePkg/Universal/Console/ConSplitterDxe` subdirectory. This driver multiplexes multiple console output and console input devices into a single virtual console device. It produces instances of the Driver Binding Protocol for the following:

- Console input device
- Console output devices
- Standard error device
- Simple pointer devices
- Absolute 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 five different drivers, but there were many common functions between the drivers, so it also saved code space to combine these five 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 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, creates one or more child handles and installs one or more I/O-related protocols. If the child handle represents a physical device, 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 `EFI_OPEN_PROTOCOL_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`, all remaining child handles are created at once.

When a bus driver opens an I/O-related protocol on the controller handle, it typically uses an open mode of `EFI_OPEN_PROTOCOL_BY_DRIVER`. However, depending on the type of bus driver, a return code of `EFI_ALREADY_STARTED` from `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.

The [figure 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 `EFI_OPEN_PROTOCOL_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** `EFI_OPEN_PROTOCOL_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 `DisconnectController()` when a request is made to stop a bus controller.

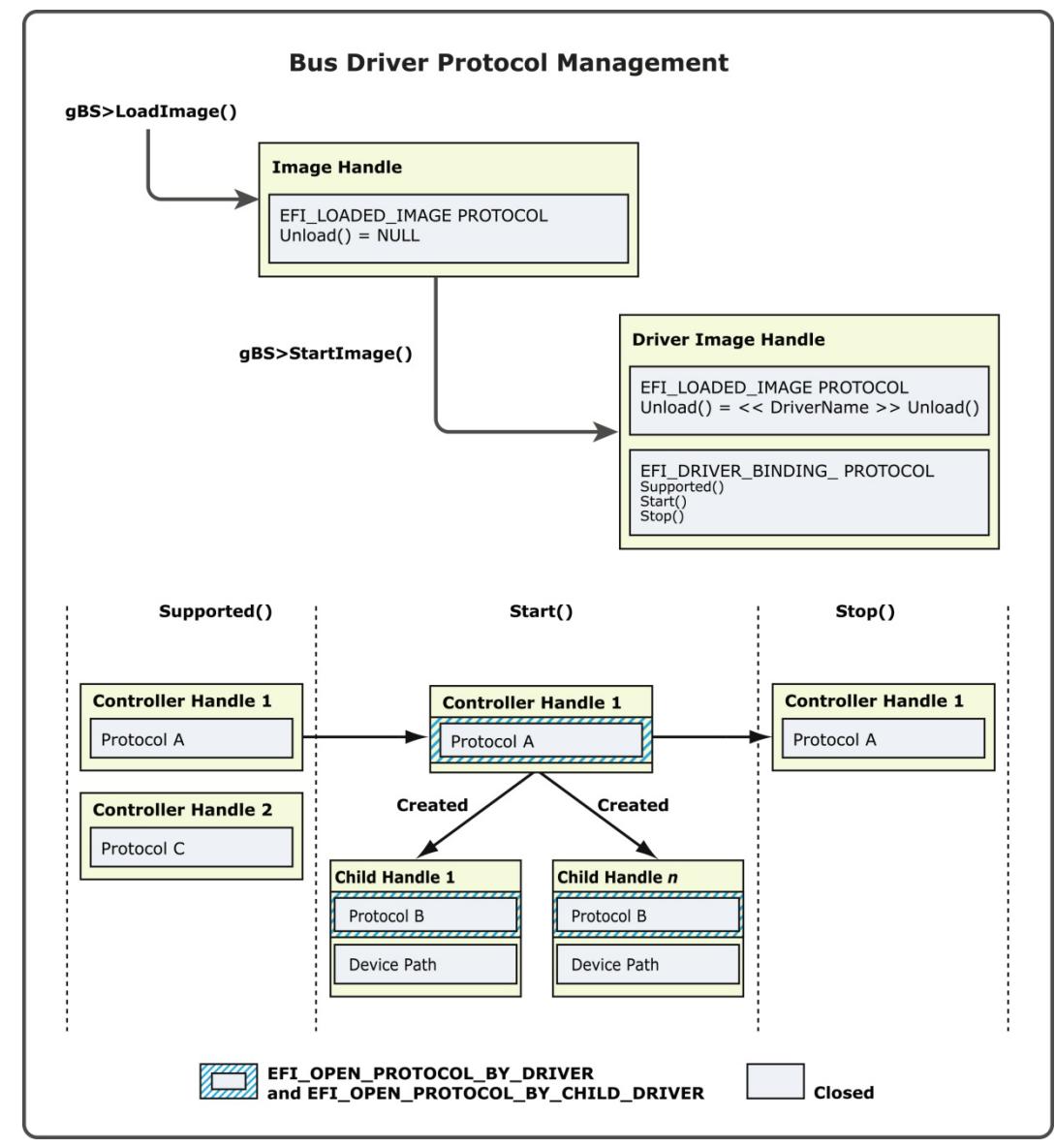


Figure 14—Bus driver protocol management

The following sections describe the subtle differences in 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`, the bus driver must produce child handles for all children. If `RemainingDevicePath` is not `NULL`, the bus driver should parse `RemainingDevicePath` and attempt to produce only the

one child device described by `RemainingDevicePath`. If the driver does not recognize the device path node in `RemainingDevicePath`, or if the device described by the device path node does not match any of the children currently attached to the bus controller, 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, a child handle should be created for that one child device.

Note: *This step does not make sense for all bus types because some 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, this ability should be implemented. Implementing this feature provides 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 `ConnectController()`, and `ConnectController()` then pass this same `RemainingDevicePath` into the `Supported()` and `Start()` services of the Driver Binding Protocol. This design allows the minimum number of drivers to be started to boot an operating system. The 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 require 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 when SCSI PUN and SCSI LUN are known ahead of time. By starting only the single mass storage boot device, on the OS required SCSI boot channel, scanning of all the other SCSI devices can be eliminated.

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()`.

6.2.8 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.

6.2.9 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 look like they were just hot added. This means that for the devices that were already plugged into the bus, the child handles are produced the first time the notification function is executed.

The USB bus driver is an example driver in the EDK II that produces no children in the `Start()` service of the Driver Binding Protocol. This driver is located at `\MdeModulePkg\Bus\Usb\UsbBusDxe` 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 actually uses the services of multiple parent controllers. This design is useful for multiplexing a group of parent controllers.

The bus driver, in this case, manages multiple parent controllers and produces a single child handle. The services produced on that single child handle 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 an example driver in the EDK II that produces children with multiple parent controllers in the `Start()` service of the Driver Binding Protocol. This driver is in the `\MdeModulePkg\Universal\Console\ConSplitterDxe` directory.

6.3 Hybrid drivers

A hybrid driver 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. 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 Extended SCSI Pass Thru Protocol (with the logical bit on) on the controller handle and creates child handles with Extended SCSI Pass Thru Protocol for each physical channel (with the logical bit off).

6.3.1 Required Hybrid Driver Features

Hybrid drivers are required to implement the following features:

- Installation of one or more instances of the [EFI_DRIVER_BINDING_PROTOCOL](#) in a driver's entry point.
- Management of one or more controller handles.

Even if a driver writer is convinced the driver manages only a single bus controller, the driver should be designed to manage multiple bus controllers. The overhead for this functionality is low, and it makes the driver more portable.

- Production of child handles.

This feature is the key distinction between device drivers and bus drivers.

- Consumption of one or more I/O-related protocols from a controller handle.
- Production of one or more I/O-related protocols on the same controller handle.
- Production of one or more I/O-related protocols on each child handle.

6.3.2 Optional Hybrid Driver Features

The following is a list of features a hybrid driver can optionally implement. Those recommended are noted below.

- Install one or more instances of the [EFI_COMPONENT_NAME2_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.

- Register one or more HII packages in the driver's entry point.

These HII packages provide strings, fonts and forms that allow users (such as IT administrators) to change the driver's configuration. These HII packages are required only if a driver must provide the ability for a user to change configuration settings for a device.

- Install one or more instances of the [EFI_DRIVER_DIAGNOSTICS2_PROTOCOL](#) in the driver's entry point.

If a driver needs to provide diagnostics for the controllers the driver manages, this protocol is required.

- Provide 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.

- Parses the `RemainingDevicePath` parameter that is passed into the `Supported()` and `Start()` services of the Driver Binding Protocol if it is not `NULL`.

This is **strongly recommended** so a bus driver can start only the one child specified by `RemainingDevicePath`. Implementing this feature may significantly improve platform boot performance.

- Install 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 the child handle represents a virtual device, then an `EFI_DEVICE_PATH_PROTOCOL` is not required.

- Install one or more instances of the `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL` in the driver's entry point.

This protocol is required only if a higher priority rule for connecting drivers to a controller through the UEFI Boot Service `ConnectController()` is needed.

- Install one or more instances of the `EFI_DRIVER_HEALTH_PROTOCOL` in the driver's entry point.

This protocol is required only for drivers that manage devices that can be in a recoverably bad state through either a repair operation or a configuration operation.

- Install one or more instances of the `EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL` in the driver's entry point.

This protocol is required only for bus drivers for a bus type where the devices on the bus can provide a container for more than one UEFI Driver. An example of such a bus type is PCI, with PCI Option ROMs on PCI Adapters containing more than one UEFI Driver.

- Install an instance of the `EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL` in the driver's entry point.

This protocol is required for PCI controller or other plug-in cards. Implementation of this feature is **recommended**.

- Create 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 a UEFI runtime driver.

6.3.2.1 Compatibility with Older EFI/UEFI Specifications

The following is the list of features a hybrid driver can optionally implement to provide compatibility with older versions of the *EFI Specification* and *UEFI Specification*.

- Install one or more instances of the `EFI_COMPONENT_NAME_PROTOCOL` in the driver's entry point.

Implementing this feature is **strongly recommended** for drivers that are required to be compatible with EFI 1.10. It allows a driver to provide human-readable names for the name of the driver and the controllers it manages. The EDK II libraries provide easy methods to produce both the Component Name Protocol and the Component Name 2 Protocol with very little additional overhead.

- Install one or more instances of the [EFI_DRIVER_CONFIGURATION_PROTOCOL](#) in the driver's entry point.

It is required if a driver must be compatible with EFI 1.10 and has any configurable options.

- Install one or more instances of the [EFI_DRIVER_CONFIGURATION2_PROTOCOL](#) in the driver's entry point.

It is required if a driver must be compatible with UEFI 2.0 and has any configurable options.

- Install one or more instances of the [EFI_DRIVER_DIAGNOSTICS_PROTOCOL](#) in the driver's entry point.

It is required if a driver must be compatible with EFI 1.10 and provide diagnostics for the controllers that the driver manages.

6.4 Service Drivers

A service driver does not manage any devices nor does it 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, leaving the driver resident in system memory. The list of features that a service driver can optionally implement follows. Recommended and optional features are noted below.

- Register one or more HII packages in the driver's entry point.

These HII packages provide strings, fonts, and forms that allow users (such as IT administrators) to change the driver's configuration. These HII packages are required only if a driver must provide the ability for a user to change configuration settings.

- Install one or more instances of the [EFI_HII_CONFIG_ACCESS_PROTOCOL](#) in the driver's entry point.

This protocol provides the services to save and restore configuration settings for a device. This protocol is required only if a driver must provide the ability for a user to change configuration settings.

- Install an instance of the [EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL](#) in the driver's entry point.

The *UEFI Specification* requires this protocol for PCI controllers or other plug-in cards. Even though this requirement does not apply to Service Drivers, implementation of this feature is still **recommended**.

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. It is the UEFI driver's responsibility to install the Device Path Protocol onto the physical controller handles because the root bridge controllers represent physical devices.

The most common example of a root bridge driver is a driver that produces the PCI Root Bridge I/O Protocol and a Device Path Protocol for each PCI Root Bridge in a platform supporting PCI.

A list of features a root bridge driver can optionally implement follows. Recommended and optional features are noted below.

- Register one or more HII packages in the driver's entry point.

These HII packages provide strings, fonts, and forms that allow users (such as IT administrators) to change the driver's configuration. These HII packages are required only if a driver must provide the ability for a user to change configuration settings for a device.

- Install one or more instances of the [EFI_HII_CONFIG_ACCESS_PROTOCOL](#) in the driver's entry point.

This protocol provides the services to save and restore configuration settings for a device. It is required only if a driver must provide the ability for a user to change configuration settings for a device.

- Install an instance of the [EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL](#) in the driver's entry point.

The *UEFI Specification* requires this protocol for PCI controllers or other plug-in cards. Even though this requirement does not apply to Root Bridge Drivers, implementation of this feature is still **recommended**.

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 currently no examples of initializing drivers in the EDK II.

7

Driver Entry Point

This chapter covers the entry point for the different categories of UEFI drivers and their optional features impacting the implementation of the driver entry point. The most common category of UEFI driver is one that follows the UEFI driver model. This category of driver is discussed first, followed by the other major types of drivers and the optional features those drivers may choose to implement. The following categories of UEFI drivers are discussed:

- UEFI Driver that follows the UEFI Driver Model
- UEFI Runtime Driver
- Initializing driver
- Root bridge driver
- Service driver

The driver entry point is the function called when a UEFI driver is started with the `StartImage()` service. At this point the driver has already been loaded into memory with the `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 `StartImage()` service transfers control to the UEFI driver at this entry point.

The [example below](#) shows the entry point to a UEFI driver called `AbcDriverEntryPoint()`. This example is expanded upon as each of UEFI driver categories and features are discussed.

The entry point to a UEFI driver is identical to the standard UEFI image entry point that is discussed in the UEFI Image Entry Point section of the *UEFI Specification*. The image handle of the UEFI driver and a pointer to the UEFI 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 UEFI system table allows the UEFI driver to make UEFI Boot Service and UEFI Runtime Service calls.

The UEFI driver can use the UEFI 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.

```
#include <Uefi.h>

/**
 This is the declaration of an EFI image entry point. This entry point
 Is the same for UEFI Applications, UEFI OS Loaders, and UEFI Drivers
 including both device drivers and bus drivers.

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

 @retval EFI_SUCCESS The operation completed successfully.
 @retval Others An unexpected error occurred.
 */
EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE     ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    return EFI_SUCCESS;
}
```

Example 86—UEFI Driver Entry Point

The name of the driver entry point function must be declared in the **[Defines]** section of the INF file associated with the UEFI Driver. The example below shows the INF file that defines **ENTRY_POINT** to the **AbcDriverEntryPoint()** function shown in the previous example. Where applicable, this INF file example is expanded upon as each of UEFI driver categories and features are discussed. See [Section 30.3](#) for more details on UEFI Driver INF files and [Appendix A](#) for a complete template of an INF file for a UEFI Driver.

```
[Defines]
INF_VERSION      = 0x00010005
BASE_NAME        = AbcDriverMinimum
FILE_GUID        = DA87D340-15C0-4824-9BF3-D52286674BEF
MODULE_TYPE      = UEFI_DRIVER
VERSION_STRING   = 1.0
ENTRY_POINT      = AbcDriverEntryPoint

[Sources]
Abc.c

[Packages]
MdePkg/MdePkg.dec

[LibraryClasses]
UefiDriverEntryPoint
```

Example 87—UEFI Driver INF File

7.1 Optional Features

This section summarizes optional features impacting the implementation of the driver entry point of a UEFI Driver. This is not a complete summary of all the optional UEFI driver features.

[Table 21](#) below provides the list of optional features and set of UEFI driver categories. If an entry in the table is empty, the feature does not ever apply to that category of UEFI Driver and must never be implemented. If the entry in the table contains a value such as 1.1 or 2.0, it means the feature optionally applies to that category of UEFI Driver if, and only if, the UEFI Driver is required to run correctly on platform firmware that conforms to that specific version of the *EFI Specification* or *UEFI Specification*.

For example, 2.0 refers to the *UEFI 2.0 Specification*, and 1.02 refers to the *EFI 1.02 Specification*. If the entry in the table contains a value followed by a '+' such as 1.1+ or 2.1+, then that means the feature optionally applies to that category of UEFI Driver if the UEFI Driver is required to run correctly on platform firmware that conforms to the version of the *EFI Specification* or *UEFI Specification* specified by the value or higher values. For example, 2.0+ refers to the *UEFI 2.0, 2.1, 2.2, 2.3, and 2.3.1 Specifications*.

This table can be used to select features that apply to a specific UEFI Driver implementation once the UEFI Driver writer knows what types of UEFI platforms with which the UEFI Driver must be compatible as well as the set of optional features the UEFI Driver must support.

For example, if a UEFI Driver is required to run on platforms that support UEFI 2.1 or higher, the Component Name Protocol, Driver Configuration Protocol, Driver Configuration 2 Protocol, and Driver Diagnostics Protocol need not be implemented because they apply only to UEFI platforms prior to UEFI 2.1.

The Driver Health Protocol may be optionally implemented, but the Driver Health Protocol is expected to be used only by a platform that is UEFI 2.2 or higher. In this case, the UEFI Driver must not depend on the Driver Health Protocol being called to function correctly because it is not called by a UEFI 2.1 platform.

Table 21—UEFI Driver Feature Selection Matrix

Feature	UEFI Driver Model				Non-UEFI Driver Model				Initializing
	Device	Bus	Hybrid	Run time	Run time	Service	Root Bridge		
Driver Binding Protocol	1.1+	1.1+	1.1+	1.1+					
Component Name 2 Protocol	2.0+	2.0+	2.0+	2.0+					
HII Packages	2.1+	2.1+	2.1+	2.1+	2.1+	2.1+	2.1+		
HII Config Access Protocol					2.1+	2.1+	2.1+		

Feature	UEFI Driver Model				Non-UEFI Driver Model			Initializing
	Device	Bus	Hybrid	Run time	Run time	Service	Root Bridge	
Driver Health Protocol	2.2+	2.2+	2.2+	2.2+				
Driver Diagnostics 2 Protocol	2.0+	2.0+	2.0+	2.0+				
Driver Family Override Protocol	2.1+	2.1+	2.1+	2.1+				
Driver Supported EFI Version Protocol	2.1+	2.1+	2.1+	2.1+	2.1+	2.1+	2.1+	
Unload()	1.02+	1.02 +	1.02+	1.02 +	1.02 +	1.02 +	1.02 +	
Exit Boot Services Event	1.02+	1.02 +	1.02+	1.02 +	1.02 +	1.02 +	1.02 +	
Set Virtual Address Map Event					1.02 +	1.02 +		
Component Name Protocol	1.1	1.1	1.1	1.1				
Driver Configuration Protocol	1.1	1.1	1.1	1.1				
Driver Configuration 2 Protocol	2.0	2.0	2.0	2.0				
Driver Diagnostics Protocol	1.1	1.1	1.1	1.1				

7.2 UEFI Driver Model

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 required to register protocol interfaces in the Handle Database and may also choose to register HII packages in the HII Database, register an `Unload()` service in the UEFI Driver's Loaded Image Protocol, and create events that are signaled when an operating system calls `ExitBootServices()` or `SetVirtualAddressMap()`. 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 UEFI boot services. The items registered in the driver entry point are used later in the boot sequence to initialize, configure, or diagnose devices required to boot an operating system.

All UEFI drivers following 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 *ImageHandle* passed into the UEFI Driver entry point. Additional instances of the Driver Binding Protocol must be installed onto new handles in the Handle Database.

The EDK II library **UefiLib** provides four functions that simplify the implementation of the driver entry point of a UEFI driver. The examples in this section make use of these driver initialization functions as shown in the following example.

```

EFI_STATUS
EFIAPI
EfiLibInstallDriverBinding (
    IN CONST EFI_HANDLE           ImageHandle,
    IN CONST EFI_SYSTEM_TABLE     *SystemTable,
    IN EFI_DRIVER_BINDING_PROTOCOL *DriverBinding,
    IN EFI_HANDLE                 DriverBindingHandle
);

EFI_STATUS
EFIAPI
EfiLibInstallAllDriverProtocols (
    IN CONST EFI_HANDLE           ImageHandle,
    IN CONST EFI_SYSTEM_TABLE     *SystemTable,
    IN EFI_DRIVER_BINDING_PROTOCOL *DriverBinding,
    IN EFI_HANDLE                 DriverBindingHandle,
    IN CONST EFI_COMPONENT_NAME_PROTOCOL *ComponentName,      OPTIONAL
    IN CONST EFI_DRIVER_CONFIGURATION_PROTOCOL *DriverConfiguration, OPTIONAL
    IN CONST EFI_DRIVER_DIAGNOSTICS_PROTOCOL *DriverDiagnostics OPTIONAL
);

EFI_STATUS
EFIAPI
EfiLibInstallDriverBindingComponentName2 (
    IN CONST EFI_HANDLE           ImageHandle,
    IN CONST EFI_SYSTEM_TABLE     *SystemTable,
    IN EFI_DRIVER_BINDING_PROTOCOL *DriverBinding,
    IN EFI_HANDLE                 DriverBindingHandle,
    IN CONST EFI_COMPONENT_NAME_PROTOCOL *ComponentName,      OPTIONAL
    IN CONST EFI_COMPONENT_NAME2_PROTOCOL *ComponentName2       OPTIONAL
);

EFI_STATUS
EFIAPI
EfiLibInstallAllDriverProtocols2 (
    IN CONST EFI_HANDLE           ImageHandle,
    IN CONST EFI_SYSTEM_TABLE     *SystemTable,
    IN EFI_DRIVER_BINDING_PROTOCOL *DriverBinding,
    IN EFI_HANDLE                 DriverBindingHandle,
    IN CONST EFI_COMPONENT_NAME_PROTOCOL *ComponentName,      OPTIONAL
    IN CONST EFI_COMPONENT_NAME2_PROTOCOL *ComponentName2,      OPTIONAL
    IN CONST EFI_DRIVER_CONFIGURATION_PROTOCOL *DriverConfiguration, OPTIONAL
    IN CONST EFI_DRIVER_CONFIGURATION2_PROTOCOL *DriverConfiguration2, OPTIONAL
    IN CONST EFI_DRIVER_DIAGNOSTICS_PROTOCOL *DriverDiagnostics, OPTIONAL
    IN CONST EFI_DRIVER_DIAGNOSTICS2_PROTOCOL *DriverDiagnostics2 OPTIONAL
);

```

Example 88—EDK II UefiLib driver initialization functions

EfiLibInstallDriverBinding() installs the Driver Binding Protocol onto the handle specified by *DriverBindingHandle*. *DriverBindingHandle* is typically the same as *ImageHandle*, but if it is **NULL**, the Driver Binding Protocol is installed onto a newly

created handle. This function is typically used by a UEFI Driver that does not implement any of the optional driver features.

`EfiLibInstallAllDriverProtocols()` installs the Driver Binding Protocol, and the driver-related protocols from the older *UEFI Specification* (and *EFI Specification*), onto the handle specified by `DriverBindingHandle`. The optional driver-related protocols are defined as `OPTIONAL` because they can be `NULL` if a driver is not producing that specific optional protocol. Once again, the `DriverBindingHandle` is typically the same as `ImageHandle`, but if it is `NULL`, all driver-related protocols are installed onto a newly created handle. This function is typically used by a UEFI Driver required to be compatible with the *EFI 1.10 Specification*.

`EfiLibInstallDriverBindingComponentName2()` installs the Driver Binding Protocol and the Component Name Protocols onto the handle specified by `DriverBindingHandle`. The optional driver-related protocols are defined as `OPTIONAL` because they can be `NULL` if a driver is not producing that specific optional protocol. Once again, the `DriverBindingHandle` is typically the same as `ImageHandle`, but if it is `NULL`, all driver-related protocols are installed onto a newly created handle. This function is typically used by a UEFI Driver that implements the Component Name Protocols that are **strongly recommended**.

`EfiLibInstallAllDriverProtocols2()` installs the Driver Binding Protocol, Component Name Protocols, Driver Configuration Protocols, and Driver Diagnostics Protocols onto the handle specified by `DriverBindingHandle`. The optional driver-related protocols are defined as `OPTIONAL` because they can be `NULL` if a driver is not producing that specific optional protocol. Once again, the `DriverBindingHandle` is typically the same as `ImageHandle`, but if it is `NULL`, all driver-related protocols are installed onto a newly created handle. This function is typically used by a UEFI Driver required to be compatible with all versions of the *UEFI Specification* and *EFI Specification*.

7.2.1 Single Driver Binding Protocol

The following is an example of the entry point to the `Abc` driver that installs the Driver Binding Protocol `gAbcDriverBinding`, the Component Name 2 Protocol `gAbcComponentName2`, 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 or features. This driver returns the status from the UEFI driver library function `EfiLibInstallDriverBindingComponentName2()`. See [Chapter 9](#) for details on the services and data fields produced by the Driver Binding Protocol and [Chapter 11](#) for details on the Component Name 2 Protocol and the Component Name Protocol

Notice that the Component Name Protocol and the Component Name 2 Protocol use the same function pointers for their services called `AbcGetDriverName()` and `AbcGetControllerName()`. This is a size reduction technique supported by the EDK II that reduces the overhead for a single UEFI Driver to support both Component Name Protocols

Also note that the optional protocol structures are declared with `GLOBAL_REMOVE_IF_UNREFERENCED`. This style allows these structures and the associated services to be removed if the Component Name feature is disabled when this UEFI driver is compiled. The EDK II library `UefiLib` uses several Platform Configuration Database (PCD) feature flags to enable and disable the Component Name Protocols and Driver Diagnostics Protocols at build time. This allows a developer to implement all of these in the UEFI Driver sources and select the ones that are actually needed for a

specific release at build time. [Chapter 30](#) covers how to build UEFI Drivers in the EDK II and also covers configuration of UEFI Drivers through PCD settings.

Note: This example and many other examples throughout this guide make use of the EDK II library `DebugLib` that provides `DEBUG()` and `ASSERT()` related macros. These macros are very useful during development and debug. However, `ASSERT()` related macros must be disabled when UEFI Drivers are released. [Chapter 31](#) covers the PCD setting to enable and disable the macros provided by `DebugLib`.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10

EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
    AbcSupported,
    AbcStart,
    AbcStop,
    ABC_VERSION,
    NULL,
    NULL
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME) AbcGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) AbcGetControllerName,
    "eng"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
    AbcGetDriverName,
    AbcGetControllerName,
    "en"
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
)
{
    EFI_STATUS  Status;

    //
    // Install driver model protocol(s) onto ImageHandle
    //
    Status = EfiLibInstallDriverBindingComponentName2 (
        ImageHandle,           // ImageHandle
        SystemTable,          // SystemTable
        NULL
    );
}
```

```

    &gAbcDriverBinding, // DriverBinding
    ImageHandle,       // DriverBindingHandle
    &gAbcComponentName, // ComponentName
    &gAbcComponentName2 // ComponentName2
);
ASSERT_EFI_ERROR (Status);

return Status;
}

```

Example 89—Single Driver Binding Protocol

The following example shows another example of the entry point to the `Abc` driver that installs the Driver Binding Protocol `gAbcDriverBinding` onto the `Abc` driver's image handle and the optional driver-related protocols. This driver returns the status from the UEFI driver library function `EfiLibInstallAllDriverProtocols2()`. This library function is used if one or more of the optional driver related protocols are being installed.

See Chapters [9](#), [11](#), [12](#), and [13](#) for details on the services and data fields produced by the Driver Binding Protocol, Component Name Protocols, Driver Configuration Protocols, and Driver Diagnostics Protocols.

```

#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>
#include <Protocol/DriverDiagnostics.h>
#include <Protocol/DriverDiagnostics2.h>
#include <Protocol/DriverConfiguration.h>
#include <Protocol/DriverConfiguration2.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10

EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
    AbcSupported,
    AbcStart,
    AbcStop,
    ABC_VERSION,
    NULL,
    NULL
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME) AbcGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) AbcGetControllerName,
    "eng"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
    AbcGetDriverName,
    AbcGetControllerName,
    "en"
};

```

```

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_CONFIGURATION_PROTOCOL gAbcDriverConfiguration = {
    (EFI_DRIVER_CONFIGURATION_SET_OPTIONS) AbcDriverConfigurationSetOptions,
    (EFI_DRIVER_CONFIGURATION_OPTIONS_VALID) AbcDriverConfigurationOptionsValid,
    (EFI_DRIVER_CONFIGURATION_FORCE_DEFAULTS) AbcDriverConfigurationForceDefaults,
    "eng"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_CONFIGURATION2_PROTOCOL gAbcDriverConfiguration2 = {
    AbcDriverConfigurationSetOptions,
    AbcDriverConfigurationOptionsValid,
    AbcDriverConfigurationForceDefaults,
    "en"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_DIAGNOSTICS_PROTOCOL gAbcDriverDiagnostics = {
    (EFI_DRIVER_DIAGNOSTICS_RUN_DIAGNOSTICS) AbcRunDiagnostics,
    "eng"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_DIAGNOSTICS2_PROTOCOL gAbcDriverDiagnostics2 = {
    AbcRunDiagnostics,
    "en"
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
)
{
    EFI_STATUS Status;

    //
    // Install driver model protocol(s) onto ImageHandle.
    //
    Status = EfiLibInstallAllDriverProtocols2 (
        ImageHandle,                               // ImageHandle
        SystemTable,                             // SystemTable
        &gAbcDriverBinding,                      // DriverBinding
        ImageHandle,                            // DriverBindingHandle
        &gAbcComponentName,                     // ComponentName2
        &gAbcComponentName2,                    // ComponentName2
        &gAbcDriverConfiguration,                // DriverConfiguration
        &gAbcDriverConfiguration2,              // DriverConfiguration2
        &gAbcDriverDiagnostics,                 // DriverDiagnostics
        &gAbcDriverDiagnostics2                // DriverDiagnostics2
    );
    ASSERT_EFI_ERROR (Status);

    return Status;
}

```

Example 90—Single Driver Binding Protocol with optional features

7.2.2

Multiple Driver Binding Protocols

If a UEFI driver supports more than one parent I/O abstraction, the driver should produce a Driver Binding Protocol for each of the parent I/O abstractions. For example, a 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 might save space and reduce maintenance. Example drivers in the EDK II 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 *ImageHandle* of the UEFI driver and additional Driver Binding Protocols are installed onto new handles. The UEFI driver library functions used in the previous two examples support the creation of new handles by passing in a **NULL** for the fourth argument. The example below shows the driver entry point for a driver that produces two instances of the Driver Binding Protocol with no optional driver-related protocols. When multiple Driver Binding Protocols are produced by a single driver, the optional driver-related protocols are installed onto the same handles as those of the Driver Binding Protocols.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10

EFI_DRIVER_BINDING_PROTOCOL gAbcFooDriverBinding = {
    AbcFooSupported,
    AbcFooStart,
    AbcFooStop,
    ABC_VERSION,
    NULL,
    NULL
};

EFI_DRIVER_BINDING_PROTOCOL gAbcBarDriverBinding = {
    AbcBarSupported,
    AbcBarStart,
    AbcBarStop,
    ABC_VERSION,
    NULL,
    NULL
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status;

    //
}
```

```

// Install first Driver Binding Protocol onto ImageHandle
//
Status = EfiLibInstallDriverBinding (
    ImageHandle,           // ImageHandle
    SystemTable,          // SystemTable
    &gAbcFooDriverBinding, // DriverBinding
    ImageHandle            // DriverBindingHandle
);
ASSERT_EFI_ERROR (Status);

//
// Install second Driver Binding Protocol onto a new handle
//
Status = EfiLibInstallDriverBinding (
    ImageHandle,           // ImageHandle
    SystemTable,          // SystemTable
    &gAbcBarDriverBinding, // DriverBinding
    NULL                  // DriverBindingHandle
);
ASSERT_EFI_ERROR (Status);

return EFI_SUCCESS;
}

```

Example 91—Multiple Driver Binding Protocols

7.2.3 Adding Driver Health Protocol Feature

The Driver Health Protocol provides services allowing a UEFI Driver to express the health status of a controller, return status messages associated with the health status, perform repair operations and request configuration changes required to place the controller in a usable state. This protocol is required only for devices that may be in a bad state which can be recovered through a repair operation or a configuration change. If a device can never be in a bad state, or a device can be in a bad state for which there is no recovery possible, this protocol should not be installed.

There are no EDK II library functions to help install the Driver Health Protocol. Instead, the UEFI Driver that requires this feature must install the Driver Health Protocol using the UEFI Boot Service `InstallMultipleProtocolInterfaces()`. Example 92, below, expands Example 91, above, and adds a Driver Health Protocol instance to `ImageHandle`, the same handle on which the Driver Binding Protocol is installed.

```

#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>
#include <Protocol/DriverHealth.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10

EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
    AbcSupported,
    AbcStart,
    AbcStop,
}

```

```

ABC_VERSION,
NULL,
NULL
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME)      AbcGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) AbcGetControllerName,
    "eng"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
    AbcGetDriverName,
    AbcGetControllerName,
    "en"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_HEALTH_PROTOCOL gAbcDriverHealth = {
    AbcGetHealthStatus,
    AbcRepair
};

EFI_STATUS
EFI API
AbcDriverEntryPoint (
    IN EFI_HANDLE          ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    EFI_STATUS  Status;

    //
    // Install driver model protocol(s) on ImageHandle
    //
    Status = EfiLibInstallDriverBindingComponentName2 (
        ImageHandle,           // ImageHandle
        SystemTable,           // SystemTable
        &gAbcDriverBinding,   // DriverBinding
        ImageHandle,           // DriverBindingHandle
        &gAbcComponentName,   // ComponentName
        &gAbcComponentName2   // ComponentName2
    );
    ASSERT_EFI_ERROR (Status);

    //
    // Install Driver Health Protocol onto ImageHandle
    //
    Status = gBS->InstallMultipleProtocolInterfaces (
        &ImageHandle,
        &gEfiDriverHealthProtocolGuid, &gAbcDriverHealth,
        NULL
    );
    ASSERT_EFI_ERROR (Status);

    return Status;
}

```

Example 92—Driver Heath Protocol Feature

7.2.4 Adding Driver Family Override Protocol Feature

The `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL` is optional. It is typically produced by UEFI Drivers associated with a set of similar controllers where multiple versions of a UEFI Driver for the set of similar controllers may be simultaneously present in a platform. This protocol allows each UEFI Driver to advertise a version number such that the UEFI Driver with the highest version is selected to manage all the controllers in the set of similar controllers.

PCI Use Case: If a platform has 3 PCI SCSI adapters from the same manufacturer, and the manufacturer requires the PCI SCSI adapter having the highest version UEFI Driver to manage all 3 PCI SCSI adapters, the Driver Family Override Protocol is required and provides the version value used to make the selection. If the Driver Family Override Protocol is not produced, the Bus Specific Driver Override Protocol for PCI selects the UEFI Driver from the adapter's PCI Option ROM to manage each adapter.

There are no EDK II library functions to help install the Driver Family Override Protocol. Instead, the UEFI Driver requiring this feature must install the Driver Family Override Protocol using the UEFI Boot Service `InstallMultipleProtocolInterfaces()`. Example 93, below, expands Example 92, above, and adds a Driver Family Override Protocol instance to `ImageHandle`, the same handle on which the Driver Binding Protocol is installed.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>
#include <Protocol/DriverFamilyOverride.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10

EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
    AbcSupported,
    AbcStart,
    AbcStop,
    ABC_VERSION,
    NULL,
    NULL
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME) AbcGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) AbcGetControllerName,
    "eng"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
    AbcGetDriverName,
```

```

    AbcGetControllerName,
    "en"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL gAbcDriverFamilyOverride = {
    AbcGetVersion
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE          ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    EFI_STATUS  Status;

    //
    // Install driver model protocol(s) on ImageHandle
    //
    Status = EfiLibInstallDriverBindingComponentName2 (
        ImageHandle,           // ImageHandle
        SystemTable,           // SystemTable
        &gAbcDriverBinding,   // DriverBinding
        ImageHandle,           // DriverBindingHandle
        &gAbcComponentName,   // ComponentName
        &gAbcComponentName2   // ComponentName2
    );
    ASSERT_EFI_ERROR (Status);

    //
    // Install Driver Family Override Protocol onto ImageHandle
    //
    Status = gBS->InstallMultipleProtocolInterfaces (
        &ImageHandle,
        &gEfiDriverFamilyOverrideProtocolGuid,
        &gAbcDriverFamilyOverride,
        NULL
    );
    ASSERT_EFI_ERROR (Status);

    return Status;
}

```

Example 93—Driver Family Override Protocol Feature

7.3 Adding the Driver Supported EFI Version Protocol Feature

This feature provides information on the version of the *UEFI Specification* to which the UEFI Driver conforms. The version information follows the same format as the version field in the EFI System Table. This feature is required for UEFI Drivers on PCI and other plug in cards.

There are no EDK II library functions to help install the Driver Supported EFI Version Protocol. Instead, the UEFI Driver requiring this feature must install the Driver Supported EFI Version Protocol using the UEFI Boot Service `InstallMultipleProtocolInterfaces()`. A UEFI Driver must install, at most, one instance of this protocol and, if it is produced, it must be installed onto the `ImageHandle`. This protocol is composed of only data fields, so no functions need be implemented to complete its implementation. Example 94, below, expands Example 93, above, and adds a Driver Supported EFI Version Protocol instance to `ImageHandle`. The Driver Supported EFI Version Protocol instance in this example specifies that this UEFI Driver follows the *UEFI 2.3.1 Specification*.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>
#include <Protocol/DriverSupportedEfiVersion.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10

EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
    AbcSupported,
    AbcStart,
    AbcStop,
    ABC_VERSION,
    NULL,
    NULL
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME) AbcGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) AbcGetControllerName,
    "eng"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
    AbcGetDriverName,
    AbcGetControllerName,
    "en"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL gAbcDriverSupportedEfiVersion =
{
    sizeof (EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL),
    EFI_2_31_SYSTEM_TABLE_REVISION
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
}
```

```

        )
{
    EFI_STATUS Status;

    //
    // Install Driver Supported EFI Version Protocol onto ImageHandle
    //
    Status = gBS->InstallMultipleProtocolInterfaces (
        &ImageHandle,
        &gEfiDriverSupportedEfiVersionProtocolGuid,
        & gAbcDriverSupportedEfiVersion,
        NULL
    );
    ASSERT_EFI_ERROR (Status);

    //
    // Install driver model protocol(s) on ImageHandle
    //
    Status = EfiLibInstallDriverBindingComponentName2 (
        ImageHandle,           // ImageHandle
        SystemTable,           // SystemTable
        &gAbcDriverBinding,   // DriverBinding
        ImageHandle,           // DriverBindingHandle
        &gAbcComponentName,   // ComponentName
        &gAbcComponentName2   // ComponentName2
    );
    ASSERT_EFI_ERROR (Status);

    return Status;
}

```

Example 94—Driver Supported EFI Version Protocol Feature

7.4 Adding HII Packages Feature

HII packages provide strings, fonts, and forms that allow users (such as IT administrators) to change the configuration of UEFI managed devices. These HII packages are required only if a driver must provide the ability for a user to change configuration settings for a device. A UEFI Driver registers HII packages in the HII Database.

The Image Services and the Human Interface Infrastructure Overview sections of the *UEFI Specification* define a method for HII packages associated with a UEFI Driver to be automatically installed as a protocol on *ImageHandle* when the UEFI Driver is loaded using the UEFI Boot Service *LoadImage()*. The HII packages are stored in a resource section of the PE/COFF image. The driver entry point of a UEFI Driver is responsible for looking up the HII Package List on *ImageHandle* and registering that list of HII packages into the HII Database. The example below shows an example of a driver entry point that performs such a registration process.

```
#include <Uefi.h>
#include <Protocol/HiiDatabase.h>
#include <Protocol/HiiPackageList.h>
```

```

#include <Library/UefiBootServicesTableLib.h>
#include <Library/DebugLib.h>

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
)
{
    EFI_STATUS                  Status;
    EFI_HII_PACKAGE_LIST_HEADER *PackageListHeader;
    EFI_HII_DATABASE_PROTOCOL   *HiiDatabase;
    EFI_HII_HANDLE              HiiHandle;

    //
    // Retrieve HII Package List Header on ImageHandle
    //
    Status = gBS->OpenProtocol (
        ImageHandle,
        &gEfiHiiPackageListProtocolGuid,
        (VOID **) &PackageListHeader,
        ImageHandle,
        NULL,
        EFI_OPEN_PROTOCOL_GET_PROTOCOL
    );
    ASSERT_EFI_ERROR (Status);

    //
    // Retrieve the pointer to the UEFI HII Database Protocol
    //
    Status = gBS->LocateProtocol (
        &gEfiHiiDatabaseProtocolGuid,
        NULL,
        (VOID **) &HiiDatabase
    );
    ASSERT_EFI_ERROR (Status);

    //
    // Register list of HII packages in the HII Database
    //
    Status = HiiDatabase->NewPackageList (
        HiiDatabase,
        PackageListHeader,
        NULL,
        &HiiHandle
    );
    ASSERT_EFI_ERROR (Status);

    return EFI_SUCCESS;
}

```

Example 95—HII Packages feature

The EDK II provides a simple way for a UEFI Driver to declare that HII packages are provided by setting `UEFI_HII_RESOURCE_SECTION` to `TRUE` in the `[Defines]` section of the INF file. This informs an EDK II build that the UEFI Driver implementation provides UNI and VFR source files that must be converted into HII packages stored in the PE/COFF

resource section of the UEFI Driver image. See [Chapter 12](#) for more details on the implementation of UNI and VFR files. The following example shows the INF file that defines `UEFI_HII_RESOURCE_SECTION` to `TRUE`. See [Section 30.3](#) for more details on UEFI Driver INF files and [Appendix A](#) for a complete template of the INF file for a UEFI Driver.

```
[Defines]
INF_VERSION = 0x00010005
BASE_NAME = AbcDriverHiiPackage
FILE_GUID = 0E474237-D123-40c2-A585-CD46279879D4
MODULE_TYPE = UEFI_DRIVER
VERSION_STRING = 1.0
ENTRY_POINT = AbcDriverEntryPoint
UEFI_HII_RESOURCE_SECTION = TRUE

[Sources]
Abc.c
AbcStrings.uni
AbcForms.vfr

[Packages]
MdePkg/MdePkg.dec

[LibraryClasses]
UefiDriverEntryPoint

[Protocols]
gEfiHiiPackageListProtocolGuid
gEfiHiiDatabaseProtocolGuid
```

Example 96—UEFI Driver INF File with HII Packages feature

7.5 Adding HII Config Access Protocol Feature

This protocol provides the services to save and restore configuration settings for a device. For drivers following the UEFI Driver Model, this protocol is typically installed in the Driver Binding Protocol `Start()` function for each device the driver manages. Only UEFI Drivers not following the UEFI Driver Model would install this protocol in the driver entry point. As a result, only the Service Drivers and Root Bridge Drivers required to save and restore configuration settings can install the HII Config Access Protocol in the driver entry point.

There are no EDK II library functions to help install the HII Config Access Protocol. Instead, the UEFI Driver requiring this feature must install the HII Config Access Protocol using the UEFI Boot Service `InstallMultipleProtocolInterfaces()`. Example 97, below, expands Example 96, above, and adds an HII Config Access Protocol instance to `ImageHandle`.

```
#include <Uefi.h>
#include <Protocol/HiiConfigAccess.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/DebugLib.h>
```

```

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_HII_CONFIG_ACCESS_PROTOCOL gAbcHiiConfigAccess = {
    AbcExtractConfig,
    AbcRouteConfig,
    AbcRouteCallback
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    EFI_STATUS  Status;

    //
    // Install HII Config Access Protocol onto ImageHandle
    //
    Status = gBS->InstallMultipleProtocolInterfaces (
        &ImageHandle,
        &gEfiHiiConfigAccessProtocolGuid,
        &gAbcHiiConfigAccess,
        NULL
    );
    ASSERT_EFI_ERROR (Status);

    return Status;
}

```

Example 97—HII Config Access Protocol Feature

7.6 Adding the Unload Feature

Any UEFI driver can be made unloadable. This feature is useful for some driver categories, but it may not be useful at all for other driver categories. It does not make sense to add the unload feature to an initializing driver because this category of driver already returns an error from the driver entry point, which forces the UEFI Image Services to automatically unload the initializing driver.

Similarly, it usually doesn't make sense for root bridge drivers or service drivers to add the unload feature. These categories of driver typically produce protocols 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, any UEFI driver using the protocols from those drivers would start to fail. If a root bridge driver or service driver guarantees that it is not being used by any other UEFI components, they may be unloaded without any adverse side effects.

Still, the `Unload()` function can be very helpful. It allows the “unload” command in the UEFI Shell to completely remove a 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 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 `DisconnectUefi`

Shell command. If a driver is just disconnected, 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, even those having the same version number, can safely be installed in the system without concern the older version of the driver may be invoked during the next connect or reconnect operation.

Be mindful that, because `Unload()` completely removes the driver from system memory, it might not be possible to load it back into the system in the same session. 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 a 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 following the UEFI driver model add the unload feature. It 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 98, below, shows the same driver entry point from [Example 89](#) (earlier in this section) with the unload feature added. Example 98 shows only a template for the `Unload()` function because the implementation of this service varies from driver to driver. The `Unload()` service is responsible for cleaning up everything the driver has done since initialization. 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 the `Unload()` service does not want to unload the driver at the time the `Unload()` service is called, it may return an error and not unload the driver. The only way a driver can actually be unloaded is by ensuring that the `Unload()` service has been registered in the `EFI_LOADED_IMAGE_PROTOCOL` and that it returns `EFI_SUCCESS`.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10

EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
    AbcSupported,
    AbcStart,
```

```

AbcStop,
ABC_VERSION,
NULL,
NULL
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME)      AbcGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) AbcGetControllerName,
    "eng"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
    AbcGetDriverName,
    AbcGetControllerName,
    "en"
};

EFI_STATUS
EFIAPI
AbcUnload (
    IN EFI_HANDLE ImageHandle
)
{
    return EFI_SUCCESS;
}

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE          ImageHandle,
    IN EFI_SYSTEM_TABLE   *SystemTable
)
{
    EFI_STATUS  Status;

    //
    // Install driver model protocol(s).
    //
    Status = EfiLibInstallDriverBindingComponentName2 (
        ImageHandle,           // ImageHandle
        SystemTable,           // SystemTable
        &gAbcDriverBinding,    // DriverBinding
        ImageHandle,           // DriverBindingHandle
        &gAbcComponentName,    // ComponentName
        &gAbcComponentName2    // ComponentName2
    );
    ASSERT_EFI_ERROR (Status);

    return Status;
}

```

Example 98—Add the Unload feature

The EDK II provides an easy method to declare the name of the UEFI driver specific `Unload()` function in the `[Defines]` section of the INF file for the UEFI Driver. Example

99, below, shows the INF file that defines `UNLOAD_IMAGE` to the `AbcUnload()` function shown in the previous example. The specified `Unload()` function automatically registers in the `EFI_LOADED_IMAGE_PROTOCOL` before the entry point of the UEFI Driver is called. See [Section 30.3](#) for more details on UEFI Driver INF files and [Appendix A](#) for a complete template of the INF file for a UEFI Driver.

```
[Defines]
INF_VERSION      = 0x00010005
BASE_NAME        = Abc
FILE_GUID        = DA87D340-15C0-4824-9BF3-D52286674BEF
MODULE_TYPE      = UEFI_DRIVER
VERSION_STRING   = 1.0
ENTRY_POINT      = AbcDriverEntryPoint
UNLOAD_IMAGE     = AbcUnload

[Sources]
Abc.c

[Packages]
MdePkg/MdePkg.dec

[LibraryClasses]
UefiDriverEntryPoint
UefiBootServicesTableLib
UefiLib
DebugLib
MemoryAllocationLib
```

Example 99—UEFI Driver INF File with Unload feature

Example 100, below, shows one possible implementation of the `Unload()` function for a UEFI driver following the UEFI driver model. It finds all the devices it manages and disconnects the driver from those devices. Next, the protocol interfaces installed in the driver entry point must be removed. The example shown here matches the driver entry point from [Example 98](#), above. 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.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>

EFI_STATUS
EFIAPI
AbcUnload (
    IN EFI_HANDLE ImageHandle
)
{
    EFI_STATUS Status;
    EFI_HANDLE *HandleBuffer;
```

```

UINTN      HandleCount;
UINTN      Index;

//
// Retrieve array of all handles in the handle database
//
Status = gBS->LocateHandleBuffer (
    AllHandles,
    NULL,
    NULL,
    &HandleCount,
    &HandleBuffer
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Disconnect the current driver from handles in the handle database
//
for (Index = 0; Index < HandleCount; Index++) {
    Status = gBS->DisconnectController (
        HandleBuffer[Index],
        gImageHandle,
        NULL
    );
}

//
// Free the array of handles
//
FreePool (HandleBuffer);

//
// Uninstall protocols installed in the driver entry point
//
Status = gBS->UninstallMultipleProtocolInterfaces (
    ImageHandle,
    &gEfiDriverBindingProtocolGuid,  &gAbcDriverBinding,
    &gEfiComponentNameProtocolGuid,  &gAbcComponentName,
    &gEfiComponentName2ProtocolGuid,  &gAbcComponentName2,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Do any additional cleanup that is required for this driver
//

return EFI_SUCCESS;
}

```

Example 100—UEFI Driver Model Unload Feature

7.7 Adding the Exit Boot Services feature

Some UEFI drivers may need to put their devices into a quiescent state or 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 a 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 a UEFI driver up to the point the OS was booted
- A device that was managed for a short period of time by a UEFI driver

In the rare case when a UEFI driver is required to place a device in a quiescent or 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 UEFI boot service `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 UEFI memory map are allowed (see the Image Services section of the *UEFI Specification*). This requirement means that the notification function for an Exit Boot Services event is not allowed to directly or indirectly allocate or free any memory through the UEFI memory services.

Examples from the EDK II that use this feature are the PCI device drivers for USB Host Controllers. Some USB Host Controllers are PCI Bus Masters that continuously access a memory buffer to poll for operation requests. Access to this memory buffer by a USB Host Controller may be required to boot an operating system, but this activity must be terminated when the OS calls `ExitBootServices()`. The typical action in the Exit Boot Services Event for these types of drivers is to disable the PCI bus master and place the USB Host Controller into a halted state

Example 101, below, shows the same example as in Example 100, above, 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 typically contains code to find the list of device handles that the driver is currently managing, and it then performs 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.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10
```

```

//
// Global variable for Exit Boot Services event.
//
EFI_EVENT mExitBootServicesEvent = NULL;

//
// Driver Binding Protocol Instance
//
EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
    AbcSupported,
    AbcStart,
    AbcStop,
    ABC_VERSION,
    NULL,
    NULL
};

//
// Component Name Protocol Instance
//
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME) AbcGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) AbcGetControllerName,
    "eng"
};

//
// Component Name 2 Protocol Instance
//
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
    AbcGetDriverName,
    AbcGetControllerName,
    "en"
};

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
AbcDriverEntryPoint (
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status;

```

```

// Create an Exit Boot Services event.
//
Status = gBS->CreateEvent (
    EVT_SIGNAL_EXIT_BOOT_SERVICES,           // Type
    TPL_NOTIFY,                            // NotifyTpl
    AbcNotifyExitBootServices,             // NotifyFunction
    NULL,                                  // NotifyContext
    &mExitBootServicesEvent                // Event
);
ASSERT_EFI_ERROR (Status);

// Install driver model protocol(s).
//
Status = EfiLibInstallDriverBindingComponentName2 (
    ImageHandle,                          // ImageHandle
    SystemTable,                           // SystemTable
    &gAbcDriverBinding,                  // DriverBinding
    ImageHandle,                          // DriverBindingHandle
    &gAbcComponentName,                 // ComponentName
    &gAbcComponentName2                // ComponentName2
);
ASSERT_EFI_ERROR (Status);

return Status;
}

```

Example 101—Adding the Exit Boot Services feature

If a UEFI driver supports both the unload feature and the Exit Boot Services feature, the `Unload()` function must close the Exit Boot Services event by calling `CloseEvent()`. This event is typically declared as a global variable so it can be easily accessed from the `Unload()` function. The following example is the same as the previous example, except the entry point looks up the `EFI_LOADED_IMAGE_PROTOCOL` associated with `ImageHandle` and registers the `Unload()` function called `AbcUnload()`. `AbcUnload()` closes the event created in the driver entry point using the UEFI Boot Service `CloseEvent()`.

```

#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10

//
// Global variable for Exit Boot Services event.
//
EFI_EVENT mExitBootServicesEvent = NULL;

//
// Driver Binding Protocol Instance
//
EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {

```

```

    AbcSupported,
    AbcStart,
    AbcStop,
    ABC_VERSION,
    NULL,
    NULL
};

/*
// Component Name Protocol Instance
//
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME)      AbcGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) AbcGetControllerName,
    "eng"
};

/*
// Component Name 2 Protocol Instance
//
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
    AbcGetDriverName,
    AbcGetControllerName,
    "en"
};

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
)
{
    EFI_STATUS Status;

    /*
    // Uninstall protocols installed in the driver entry point
    //
    Status = gBS->UninstallMultipleProtocolInterfaces (
        ImageHandle,
        &gEfiDriverBindingProtocolGuid,  &gAbcDriverBinding,
        &gEfiComponentNameProtocolGuid, &gAbcComponentName,
        &gEfiComponentName2ProtocolGuid, &gAbcComponentName2,
        NULL
    );
}

```

```

if (EFI_ERROR (Status)) {
    return Status;
}

//
// Close Exit Boot Services event created in the driver entry point
//
gBS->CloseEvent (mExitBootServicesEvent);

return EFI_SUCCESS;
}

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
)
{
    EFI_STATUS             Status;

    //
    // Create an Exit Boot Services event.
    //
    Status = gBS->CreateEvent (
        EVT_SIGNAL_EXIT_BOOT_SERVICES,      // Type
        TPL_NOTIFY,                         // NotifyTpl
        AbcNotifyExitBootServices,          // NotifyFunction
        NULL,                             // NotifyContext
        &mExitBootServicesEvent            // Event
    );
    ASSERT_EFI_ERROR (Status);

    //
    // Install driver model protocol(s).
    //
    Status = EfiLibInstallDriverBindingComponentName2 (
        ImageHandle,                      // ImageHandle
        SystemTable,                      // SystemTable
        &gAbcDriverBinding,               // DriverBinding
        ImageHandle,                      // DriverBindingHandle
        &gAbcComponentName,              // ComponentName
        &gAbcComponentName2              // ComponentName2
    );
    ASSERT_EFI_ERROR (Status);

    return Status;
}

```

Example 102—Add the Unload and Exit Boot Services event features

7.8 Initializing Driver entry point

The [example below](#) shows 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 returns an error

from the entry point so the driver is unloaded by the UEFI image services. An initializing driver never registers an `Unload()` service because an initializing driver is always unloaded after the driver entry point is executed. This type is typically used by OEMs and IBVs to initialize the state of a hardware component in the platform such as a processor or chipset component.

```
#include <Uefi.h>

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    //
    // Perform some platform initialization operations here
    //

    return EFI_ABORTED;
}
```

Example 103—Initializing driver entry point

7.9 Service Driver entry point

A service driver produces one or more protocol interfaces on the driver's image handle or on newly created handles. The example 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.

Caution: *The `Unload()` service for a service driver may 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 UEFI components. If the service driver is unloaded and other UEFI components are still using the protocols that were produced by the unloaded driver, then the system is likely to fail.*

```
#include <Uefi.h>
#include <Protocol/Decompress.h>
#include <Library/UefiBootServicesTableLib.h>

//
// Decompress Protocol instance
//
EFI_DECOMPRESS_PROTOCOL gAbcDecompress = {
    AbcGetInfo,
    AbcDecompress
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
```

```

    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
//
// Install Decompress Protocol onto UEFI Driver's ImageHandle
//
return gBS->InstallMultipleProtocolInterfaces (
    &ImageHandle,
    &gEfiDecompressProtocolGuid, &gAbcDecompress,
    NULL
);
}

```

Example 104—Service driver entry point using image handle

A service driver may also install its protocol interfaces onto one or more new handles in the Handle Database. The following example shows a template for a service driver called **Abc** that produces the Decompress Protocol on a new handle.

```

#include <Uefi.h>
#include <Protocol/Decompress.h>
#include <Library/UefiBootServicesTableLib.h>

//
// Handle for the Decompress Protocol
//
EFI_HANDLE gAbcDecompressHandle = NULL;

//
// Decompress Protocol instance
//
EFI_DECOMPRESS_PROTOCOL gAbcDecompress = {
    AbcGetInfo,
    AbcDecompress
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
//
// Install Decompress Protocol onto a new handle
//
return gBS->InstallMultipleProtocolInterfaces (
    &gAbcDecompressHandle,
    &gEfiDecompressProtocolGuid, &gAbcDecompress,
    NULL
);
}

```

Example 105—Service driver entry point creating new handle

7.10 Root bridge driver entry point

Root bridge drivers produce handles and software abstractions for the bus types directly produced by a core chipset. The PCI Root Bridge I/O Protocol is an example of a software abstraction for root bridges that is defined in the PCI Bus Support chapter of the *UEFI Specification*.

UEFI drivers that produce root bridge abstractions 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 creates new handles.
- It installs a software abstraction for each root bridge, such as the PCI Root Bridge I/O Protocol
- It installs a Device Path Protocol for each root bridge that describes the programmatic path to the root bridge device.

A root bridge driver may register an `Unload()` service, and that service would be required to uninstall the protocols that were installed in the driver's entry point.

Caution: *The `Unload()` service for a root bridge driver may be a dangerous operation because there is no way for the root bridge driver to know if the protocols it installed are being used by other UEFI components. If the root bridge driver is unloaded and other UEFI components are still using the protocols that were produced by the unloaded driver, then the system is likely to fail.*

The example, below shows an example of 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. Additional private data may need to be required to properly manage a PCI root bridge.

```
#include <Uefi.h>
#include <Protocol/DevicePath.h>
#include <Protocol/PciRootBridgeIo.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/DevicePathLib.h>

// 
// Structure defintion for the device path of a PCI Root Bridge
//
typedef struct {
    ACPI_HID_DEVICE_PATH      AcpiDevicePath;
    EFI_DEVICE_PATH_PROTOCOL  EndDevicePath;
} EFI_PCI_ROOT_BRIDGE_DEVICE_PATH;
```

```

// Handle for the PCI Root Bridge
//
EFI_HANDLE gAbcPciRootBridgeIoHandle = NULL;

//
// Device Path Protocol instance for the PCI Root Bridge
//
EFI_PCI_ROOT_BRIDGE_DEVICE_PATH gAbcPciRootBridgeIoDevicePath = {
{
    ACPI_DEVICE_PATH,                                // Type
    ACPI_DP,                                         // Subtype
    (UINT8)(sizeof(ACPI_HID_DEVICE_PATH)),           // Length lower
    (UINT8)((sizeof(ACPI_HID_DEVICE_PATH)) >> 8), // Length upper
    EISA_PNP_ID(0x0A03),                            // HID
    0                                                 // UID
},
{
    END_DEVICE_PATH_TYPE,                           // Type
    END_ENTIRE_DEVICE_PATH_SUBTYPE, // Subtype
    END_DEVICE_PATH_LENGTH,                         // Length
    0                                              // Length
},
};

//
// PCI Root Bridge I/O Protocol instance for the PCI Root Bridge
//
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL gAbcPciRootBridgeIo = {
    NULL,                                         // ParentHandle
    AbcPciRootBridgeIoPollMem,                     // PollMem()
    AbcPciRootBridgeIoPolIo,                       // 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_STATUS
EFIAPI
AbcDriverEntryPoint (

```

```

    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
//
// Perform PCI Root Bridge initialization operations here
//

//
// Install the Device Path Protocol and PCI Root Bridge I/O Protocol
// onto a new handle.
//
return gBS->InstallMultipleProtocolInterfaces (
    &gAbcPciRootBridgeIoHandle,
    &gEfiDevicePathProtocolGuid,
    &gAbcPciRootBridgeIoDevicePath,
    &gEfiPciRootBridgeIoProtocolGuid,
    &gAbcPciRootBridgeIo,
    NULL
);
}

```

Example 106—Single PCI root bridge driver entry point

The example below, shows an example 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 describing the same root bridge controller.

Templates for the Device Path Protocol and PCI Root Bridge I/O Protocol are declared as global variables, and copies of those global variable template are made for each PCI root bridge using the `AllocateCopyPool()` function in the EDK II library `MemoryAllocationLib`. Additional private data may need to be required to properly manage a group of PCI root bridges.

```

#include <Uefi.h>
#include <Protocol/DevicePath.h>
#include <Protocol/PciRootBridgeIo.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>
#include <Library/DevicePathLib.h>
#include <Library/DebugLib.h>

//
// Structure defintion for the device path of a PCI Root Bridge
//
typedef struct {
    ACPI_HID_DEVICE_PATH      AcpiDevicePath;
    EFI_DEVICE_PATH_PROTOCOL   EndDevicePath;
} EFI_PCI_ROOT_BRIDGE_DEVICE_PATH;

//
// Device Path Protocol instance for the PCI Root Bridge

```

```

// 
EFI_PCI_ROOT_BRIDGE_DEVICE_PATH gAbcPciRootBridgeIoDevicePathTemplate = {
{
    ACPI_DEVICE_PATH,                                // Type
    ACPI_DP,                                         // Subtype
    (UINT8)(sizeof(ACPI_HID_DEVICE_PATH)),           // Length lower
    (UINT8)((sizeof(ACPI_HID_DEVICE_PATH)) >> 8),   // Length upper
    EISA_PNP_ID(0x0A03),                            // HID
    0                                                 // UID
},
{
    END_DEVICE_PATH_TYPE,                           // Type
    END_ENTIRE_DEVICE_PATH_SUBTYPE,                // Subtype
    END_DEVICE_PATH_LENGTH,                         // Length
    0                                                 // Length
}
};

// 
// PCI Root Bridge I/O Protocol instance for the PCI Root Bridge
// 
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL gAbcPciRootBridgeIoTemplate = {
NULL,                                              // ParentHandle
AbcPciRootBridgeIoPollMem,                          // PollMem()
AbcPciRootBridgeIoPolIo,                           // 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_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;

//
// Perform PCI Root Bridge initialization operations here
//

for (Index = 0; Index < 4; Index++) {
    //
    // Allocate and initialize Device Path Protocol
    //
    DevicePath = AllocateCopyPool (
        sizeof (gAbcPciRootBridgeIoDevicePathTemplate),
        &gAbcPciRootBridgeIoDevicePathTemplate
    );
    ASSERT (DevicePath != NULL);
    DevicePath->AcpiDevicePath.UID = (UINT32)Index;

    //
    // Allocate and initialize PCI Root Bridge I/O Protocol
    //
    PciRootBridgeIo = AllocateCopyPool (
        sizeof (gAbcPciRootBridgeIoTemplate),
        &gAbcPciRootBridgeIoTemplate
    );
    ASSERT (PciRootBridgeIo != NULL);

    //
    // Install the Device Path Protocol and PCI Root Bridge I/O Protocol
    // onto a new handle.
    //
    NewHandle = NULL;
    Status = gBS->InstallMultipleProtocolInterfaces (
        &NewHandle,
        &gEfiDevicePathProtocolGuid,     DevicePath,
        &gEfiPciRootBridgeIoProtocolGuid, PciRootBridgeIo,
        NULL
    );
    ASSERT_EFI_ERROR (Status);
}
return EFI_SUCCESS;
}

```

Example 107—Multiple PCI root bridge driver entry point

7.11 Runtime Drivers

UEFI Runtime Drivers are not common. If a UEFI Driver does not need to provide services after `ExitBootServices()`, the UEFI Driver should not use the techniques described in this section. The best example of a runtime driver following the UEFI driver model is an UNDI driver providing services for a network interface controller (NIC).

A UEFI Runtime Driver provides services that are available after `ExitBootServices()` has been called. UEFI Drivers of this category are much more difficult to implement and validate because they are required to execute in both the pre-boot environment, where

the system firmware owns the platform, and in the post-boot environment, where an operating system owns the platform.

An OS may choose to execute in a virtual addressing mode and, as a result, may prefer to call firmware services provided by UEFI Runtime Drivers in a virtual addressing mode. A UEFI Runtime Driver must not make any assumptions about the type of operating system to be booted, so the driver must always be able to switch from using physical addresses to using virtual addresses if the operating system calls `SetVirtualAddressMap()`.

In addition, because all memory regions marked as boot services memory in the UEFI memory map are converted to available memory when the OS boots, a UEFI Runtime Driver must allocate memory buffers required by the services provided after `ExitBootServices()` in order to be allocated from runtime memory.

A UEFI Runtime Driver typically creates the following two events so the driver is notified when these important transitions occur:

- Exit Boot Services event
- Set Virtual Address Map event

The Exit Boot Services event is signaled when the OS loader or OS kernel calls `ExitBootServices()`. After this point, the UEFI driver is not allowed to use any of the UEFI boot services. The UEFI 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 `SetVirtualAddressMap()`. If this event is signaled, the OS loader or OS kernel requests that all runtime components be converted from their physical address mapping to the virtual address mappings that are then passed to `SetVirtualAddressMap()`.

The UEFI firmware below the UEFI Driver performs 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 UEFI firmware below the UEFI Driver is not aware of runtime memory buffers have been allocated by a UEFI Runtime Driver. UEFI firmware below the UEFI Driver is also not aware if there are any pointer values within those allocated buffers that must be converted from physical addresses to virtual addresses.

Caution: *The notification function for the Set Virtual Address Map event is required to use the UEFI Runtime Service `ConvertPointer()` to convert all pointers in global variables and allocated runtime buffers from physical address to virtual addresses. This code may be complex and difficult to get correct because, at this time, no tools are available to help know when all the pointers have been converted. When not done correctly, the only symptom noticed may be that the OS crashes or hangs due to a condition in the middle of a call to a service produced by a runtime driver.*

Note: *The algorithm to convert pointers can be especially complex if the UEFI Runtime Driver manages linked lists or nested structures. The `SetVirtualAddressMap()` event executes in physical mode, so all linked list and structure traversals must be performed with the physical versions of the pointer values. Once a pointer value is converted from a physical address to a virtual address, that pointer value cannot be used again within the `SetVirtualAddressMap()` event. The typical approach is to convert the pointers to the leaf structures first and work towards the root.*

The following example shows the driver entry point for a UEFI Runtime Driver that creates an Exit Boot Services event and a Set Virtual Address Map event. These events are typically declared as global variables. The notification function for the Exit Boot Services event sets a global variable `gAtRuntime` to `TRUE`, allowing the code in other functions to know if the UEFI boot services are available or not. 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 using a the `EfiConvertPointer()` function from the EDK II library `UefiRuntimeLib`. A real driver might have many more pointers to convert. In general, a UEFI Runtime Driver should be designed to reduce or eliminate pointers that need to be converted to minimize the likelihood of missing a pointer conversion.

```
#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiRuntimeLib.h>
#include <Library/DebugLib.h>

//
// Global variable for Exit Boot Services event.
//
EFI_EVENT mExitBootServicesEvent = NULL;

//
// Global variable for Set Virtual Address Map event.
//
EFI_EVENT mSetVirtualAddressMapEvent = NULL;

//
// Global variable updated when Exit Boot Services is signaled.
//
BOOLEAN gAtRuntime = FALSE;

//
// Global pointer that is converted to a virtual address when
// Set Virtual Address Map is signaled.
//
VOID *gGlobalPointer;

VOID
EFIAPI
AbcNotifyExitBootServices (
    IN EFI_EVENT Event,
    IN VOID     *Context
)
{
    gAtRuntime = TRUE;
}

VOID
EFIAPI
AbcNotifySetVirtualAddressMap (
    IN EFI_EVENT Event,
    IN VOID     *Context
)
{
    EFI_STATUS Status;
```

```

Status = EfiConvertPointer (
    EFI_OPTIONAL_PTR,
    (VOID **)&gGlobalPointer
);
}

EFI_STATUS
EFI API
AbcDriverEntryPoint (
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status;

    //
    // Create an Exit Boot Services event.
    //
    Status = gBS->CreateEvent (
        EVT_SIGNAL_EXIT_BOOT_SERVICES,           // Type
        TPL_NOTIFY,                            // NotifyTpl
        AbcNotifyExitBootServices,              // NotifyFunction
        NULL,                                  // NotifyContext
        &mExitBootServicesEvent                // Event
    );
    ASSERT_EFI_ERROR (Status);

    //
    // Create a Set Virtual Address Map event.
    //
    Status = gBS->CreateEvent (
        EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE,     // Type
        TPL_NOTIFY,                            // NotifyTpl
        AbcNotifySetVirtualAddressMap,         // NotifyFunction
        NULL,                                  // NotifyContext
        &mSetVirtualAddressMapEvent            // Event
    );
    ASSERT_EFI_ERROR (Status);

    //
    // Perform additional driver initialization here
    //

    return EFI_SUCCESS;
}

```

Example 108—UEFI Runtime Driver entry point

A UEFI Runtime Driver must have the required subsystem type in the PE/COFF image for the UEFI Boot Service `LoadImage()` to allocate memory for the code and data sections from runtime memory. In the EDK II this is done by setting `MODULE_TYPE` in the `[Defines]` section of the INF file to `DXE_RUNTIME_DRIVER`. In addition, a `MODULE_TYPE` of `DXE_RUNTIME_DRIVER` is required to have a `[Depex]` section in the INF file. UEFI Runtime Driver must use the same `[Depex]` section contents. The [example below](#) shows the INF file for a UEFI Runtime Driver with a `MODULE_TYPE` of `DXE_RUNTIME_DRIVER` and the required `[Depex]` section.

```

[Defines]
INF_VERSION      = 0x00010005
BASE_NAME        = AbcRuntimeDriver
FILE_GUID        = D3A3F14B-8ED4-438C-B7B7-FAF3F639B160
MODULE_TYPE      = DXE_RUNTIME_DRIVER
VERSION_STRING   = 1.0
ENTRY_POINT      = AbcDriverEntryPoint

[Sources]
Abc.c

[Packages]
MdePkg/MdePkg.dec

[LibraryClasses]
UefiDriverEntryPoint
UefiBootServicesTableLib
UefiRuntimeLib
DebugLib

[Depex]
gEfiBdsArchProtocolGuid          AND
gEfiCpuArchProtocolGuid          AND
gEfiMetronomeArchProtocolGuid    AND
gEfiMonotonicCounterArchProtocolGuid AND
gEfiRealTimeClockArchProtocolGuid AND
gEfiResetArchProtocolGuid        AND
gEfiRuntimeArchProtocolGuid      AND
gEfiSecurityArchProtocolGuid    AND
gEfiTimerArchProtocolGuid       AND
gEfiVariableWriteArchProtocolGuid AND
gEfiVariableArchProtocolGuid    AND
gEfiWatchdogTimerArchProtocolGuid AND

```

Example 109—UEFI Runtime Driver INF File

If a UEFI Runtime Driver also supports the unload feature, the `Unload()` function must close the Exit Boot Services and Set Virtual Address Map events by calling the UEFI Boot Service `CloseEvent()`. These events are typically declared as global variables so they can be easily accessed from the `Unload()` function. The example below shows an unloadable runtime driver. It is the same as the previous example, except the entry point looks up the `EFI_LOADED_IMAGE_PROTOCOL` associated with `ImageHandle` and registers the `Unload()` function called `AbcUnload()`. `AbcUnload()` closes the events that were created in the driver entry point using the UEFI Boot Service `CloseEvent()`.

```

#include <Uefi.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiRuntimeLib.h>
#include <Library/DebugLib.h>

//
// Global variable for Exit Boot Services event.
//
EFI_EVENT mExitBootServicesEvent = NULL;

//
// Global variable for Set Virtual Address Map event.

```

```

//  

EFI_EVENT mSetVirtualAddressMapEvent = NULL;  

//  

// Global variable updated when Exit Boot Services is signaled.  

//  

BOOLEAN gAtRuntime = FALSE;  

//  

// Global pointer that is converted to a virtual address when  

// Set Virtual Address Map is signaled.  

//  

VOID *gGlobalPointer;  

VOID  

EFIAPI  

AbcNotifyExitBootServices (
    IN EFI_EVENT Event,
    IN VOID *Context
)
{
    gAtRuntime = TRUE;
}

VOID  

EFIAPI  

AbcNotifySetVirtualAddressMap (
    IN EFI_EVENT Event,
    IN VOID *Context
)
{
    EFI_STATUS Status;  

    Status = EfiConvertPointer (
        EFI OPTIONAL_PTR,
        (VOID **) &gGlobalPointer
    );
}

EFI_STATUS  

EFIAPI  

AbcUnload (
    IN EFI_HANDLE ImageHandle
)
{
    gBS->CloseEvent (mExitBootServicesEvent);
    gBS->CloseEvent (mSetVirtualAddressMapEvent);
    return EFI_SUCCESS;
}

EFI_STATUS  

EFIAPI  

AbcDriverEntryPoint (
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status;

```

```
//  
// Create an Exit Boot Services event.  
//  
Status = gBS->CreateEvent (  
    EVT_SIGNAL_EXIT_BOOT_SERVICES, // Type  
    TPL_NOTIFY, // NotifyTpl  
    AbcNotifyExitBootServices, // NotifyFunction  
    NULL, // NotifyContext  
    &mExitBootServicesEvent // Event  
);  
ASSERT_EFI_ERROR (Status);  
  
//  
// Create a Set Virtual Address Map event.  
//  
Status = gBS->CreateEvent (  
    EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE, // Type  
    TPL_NOTIFY, // NotifyTpl  
    AbcNotifySetVirtualAddressMap, // NotifyFunction  
    NULL, // NotifyContext  
    &mSetVirtualAddressMapEvent // Event  
);  
ASSERT_EFI_ERROR (Status);  
  
//  
// Perform additional driver initialization here  
//  
  
return EFI_SUCCESS;  
}
```

Example 110—UEFI Runtime Driver entry point with Unload feature

8

Private Context Data Structures

UEFI drivers managing more than one controller need to be designed with reentrancy in mind. This means that global variables should not be used to track information about individual controllers. Instead, data structures should be allocated with the UEFI 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 introduces some object-oriented programming techniques that can be applied to drivers managing controllers. These techniques can simplify driver design and implementation. The concept of a private context data structure containing all the information required to manage a controller is introduced. This data structure contains the public data fields, public services, private data fields, and private services a UEFI driver may require to manage a controller.

Some categories of UEFI drivers do not require the use of these data structures. If a UEFI driver only produces a single protocol, or it manages, at most, one device, the techniques presented here are not required. An initializing driver does not produce any services and does not manage any devices, so it does not use this technique. A service driver that produces a single protocol and does not manage any devices does not likely use this technique. A root bridge driver that manages a single root bridge device does not likely use this technique, but a root bridge driver that manages more than one root bridge device may 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 manages only a single device in a platform, this technique should still be used because it simplifies 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.

Implementations of Hybrid drivers that follow the UEFI Driver Model may define two different private context data structures—one for the bus controller and another one for the child controllers it produces.

It is possible to use other techniques to track the information required to manage multiple controllers in a re-entrant-safe manner, but those techniques likely require more overhead in the driver itself to manage this information. The techniques presented here are intended to produce small driver executables. These techniques are used throughout the drivers in EDK II.

8.1 Containing Record Macro

The containing record macro, 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. Protocol producing UEFI drivers 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*.

The example below is the definition of the `CR()` macro from the EDK II library `DebugLib`. 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. `BASE_CR()` returns a pointer to the data structure that contains the specified field. For debug builds, `CR()` also does an additional check to verify a signature value. If the signature value does not match, an `ASSERT()` message is generated and the system is halted or generates a breakpoint. For production builds, the signature checks are typically disabled. Most UEFI drivers define additional macros based on the `CR()` macro that retrieves the private context data structure based on a *this* pointer to a produced protocol. These additional macros are typically given names that make it easier to understand in the source code that the *this* pointer is being used to retrieve the private context data structure defined by the UEFI Driver.

```
/***
Macro that calls DebugAssert() if the containing record does not have a
matching signature. If the signatures matches, then a pointer to the
data structure that contains a specified field of that data structure
is returned. This is a lightweight method that hides information by
placing a public data structure inside a larger private data structure
and using a pointer to the public data structure to retrieve a pointer
to the private data structure.

If the data type specified by TYPE does not contain the field specified
by Field, then the module will not compile.

If TYPE does not contain a field called Signature, then the module will
not compile.

@param Record      The pointer to the field specified by Field
                   within a data structure of type TYPE.

@param TYPE        The name of the data structure type to return
                   This data structure must contain the field
                   specified by Field.

@param Field       The name of the field in the data structure
                   specified by TYPE to which Record points.

@param TestSignature The 32-bit signature value to match.

*/
#ifndef !defined(MDEPKG_NDEBUG)
```

```

#define CR(Record, TYPE, Field, TestSignature) \
    (DebugAssertEnabled () && \
     (BASE_CR (Record, TYPE, Field)->Signature != TestSignature)) ? \
     (TYPE *) (_ASSERT (CR has Bad Signature), Record) : \
     BASE_CR (Record, TYPE, Field)
#else
#define CR(Record, TYPE, Field, TestSignature) \
    BASE_CR (Record, TYPE, Field)
#endif

```

Example 111—Containing record macro definitions

The following example shows the definition of the `BASE_CR()` macro from the EDK II that is used to implement the `CR()` macro above. The `BASE_CR()` macro does not perform any signature checking or handle any error conditions. This macro may be used with data structures that do not have a *Signature* field.

```

/** 
 * Macro that returns a pointer to the data structure that contains a
 * specified field of that data structure. This is a lightweight method
 * to hide information by placing a public data structure inside a larger
 * private data structure and using a pointer to the public data structure
 * to retrieve a pointer to the private data structure.
 *
 * This function computes the offset, in bytes, of field specified by
 * Field from the beginning of the data structure specified by TYPE.
 * This offset is subtracted from Record, and is used to return a pointer
 * to a data structure of the type specified by TYPE. If the data type
 * specified by TYPE does not contain the field specified by Field, then
 * the module will not compile.
 *
 * @param Record   Pointer to the field specified by Field within a data
 *                 structure of type TYPE.
 * @param TYPE     The name of the data structure type to return. This
 *                 data structure must contain the field specified by
 *                 Field.
 * @param Field    The name of the field in the data structure specified
 *                 by TYPE to which Record points.
 *
 * @return A pointer to the structure from one of its elements.
 */
#define BASE_CR(Record, TYPE, Field) \
    ((TYPE *) ((CHAR8 *) (Record) - (CHAR8 *) &((TYPE *) 0)->Field)))

```

Example 112—Containing record macro definitions

8.2 Data structure design

Proper data structure design is one of the keys to making UEFI Drivers both simple and easy to maintain. If a UEFI 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 UEFI services. By designing-in the proper fields, these complex algorithms are avoided, resulting in a driver with a smaller executable footprint.

Static data, commonly accessed data, and services related to the management of a device should all be placed in a private context data structure.

Another key requirement is that the private context data structure must be easy to find when an I/O service produced by the driver is called. The I/O services produced by a driver are exported through protocol interfaces, and all protocol interfaces include a *This* parameter as the first argument. The *This* parameter is a pointer to the protocol interface containing the I/O service being called. The data structure design presented here shows how the *This* pointer passed into an I/O service can be used to easily gain access to the private context data structure.

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. Signatures are composed of four ASCII characters in a data field of type `UINTN` and must be the first field of the structure with the field name of *Signature*. When memory dumps are performed, signatures stand out by making the beginning of specific data structures easy to identify. 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()` may be generated. If one of these `ASSERT()` messages is observed, a UEFI driver was likely passed in a bad or corrupt *This* pointer or the contents of the data structure that *This* refers too has been corrupted.

Device drivers 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 needed during I/O operations or driver-related operations. Root bridge drivers and bus drivers typically store the handle of the child that was created, and a hybrid driver typically stores both the handle of the bus controller and the handle of the child controller produced.

The group of consumed protocol interfaces is the 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 the UEFI Boot Service `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 the UEFI Boot Service `CloseProtocol()`. Basically, the stop section should mirror the start section of the driver, closing all protocols that were started.

The group of produced protocol interfaces declares the storage for the protocols that the driver produces. These protocols typically provide software abstractions for consoles or boot devices.

The number and type of 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 mass storage device may store information about the characteristics of the mass storage device such as the number of

cylinders, number of heads, and number of sectors on the physical mass storage device managed by the driver.

[Appendix A](#) contains the generic template for the `<>DriverName>.h` file with the declaration of a private context data structure that can be used for root bridge drivers, device drivers, bus drivers, or hybrid drivers. The `#define` statement above the private context data structure declaration using the `SIGNATURE_32()` macro 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.

A set of macros below the private context data structure declaration help retrieve a pointer to the private context data structure from a `This` pointer for each of the produced protocols using the `CR()` macro introduced above. These macros are the simple mechanisms that allow private data fields to be accessed from the services in each of the produced protocols.

The example below shows an example of the private context data structure from the `DiskIoDxe` driver in the `MdeModulePkg`. It contains the `#define` statement for the data structure's signature. In this case, the signature is the ASCII string "`dskI`". The example also contains a pointer to the only protocol that this driver consumes; the Block I/O Protocol. It contains storage for the only protocol this driver produces; the Disk I/O Protocol. It does not have any additional private data fields. The macro at the bottom retrieves the private context data structure from a pointer to the field called `DiskIo` that is a pointer to the one protocol that this driver produces.

```
#define DISK_IO_PRIVATE_DATA_SIGNATURE SIGNATURE_32 ('d','s','k','I')

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)
```

Example 113—Simple private context data structure

The [following example](#) shows a more complex private context data structure from the `EhciDxe` driver in the `MdeModulePkg` that manages PCI EHCI controllers and produces USB Host Controller 2 Protocols. It contains the `Signature` field that is set to "`ehci`". It also contains pointers to the consumed protocol; the PCI I/O Protocol, and storage for the USB Host Controller 2 Protocol that is produced by this driver. In addition, there are a large number of private data fields that are used during initialization and all supported USB transaction types. Details on how these private fields are used can be found in the source code to the EHCI driver in EDK II.

```

#define USB2_HC_DEV_SIGNATURE  SIGNATURE_32 ('e', 'h', 'c', 'i')

typedef struct {
    UINTN                     Signature;
    EFI_USB2_HC_PROTOCOL     Usb2Hc;
    EFI_PCI_IO_PROTOCOL     *PciIo;
    UINT64                   OriginalPciAttributes;
    USBHC_MEM_POOL          *MemPool;
    EHC_QTD                 *ShortReadStop;
    EFI_EVENT                PollTimer;
    EFI_EVENT                ExitBootServiceEvent;
    EHC_QH                  *ReclaimHead;
    VOID                     *PeriodFrame;
    VOID                     *PeriodFrameHost;
    VOID                     *PeriodFrameMap;
    EHC_QH                  *PeriodOne;
    LIST_ENTRY               AsyncIntTransfers;
    UINT32                   HcStructParams;
    UINT32                   HcCapParams;
    UINT32                   CapLen;
    EFI_UNICODE_STRING_TABLE *ControllerNameTable;
} USB2_HC_DEV;

#define EHC_FROM_THIS(a) \
CR(a, USB2_HC_DEV, Usb2Hc, USB2_HC_DEV_SIGNATURE)

```

Example 114—Complex private context data structure

8.3 Allocating private context data structures

Private context data structures are allocated in the `start()` function of the Driver Binding Protocol. The service that is typically used to allocate the private context data structures is the UEFI Boot Service `AllocatePool()`. The following example shows the generic template for allocating and zeroing a private context data structure in the `Start()` function of the Driver Binding Protocol. In this example, the UEFI Boot Service `SetMem()` is used to fill the allocated buffer with zeros. This code example shows only a fragment from the `start()` function. [Chapter 9](#) of this guide covers the services that are produced by the Driver Binding Protocol in more detail. The code examples that follow show how the implementation of `start()` can be simplified by using the EDK II library `MemoryAllocationLib`.

```

#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/DevicePath.h>
#include <Library/UefiBootServicesTableLib.h>

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 115—Allocation of a private context data structure

The example below shows the same generic template for the `Start()` function above except that it uses the EDK II library `MemoryAllocationLib` to allocate and zero the private context data structure.

```

#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/DevicePath.h>
#include <Library/MemoryAllocationLib.h>

EFI_STATUS
EFIAPI
<<DriverName>>DriverBindingStart (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
)
{
    <<DRIVER_NAME>>_PRIVATE_DATA Private;

    //
    // Allocate and zero the private context data structure
    //
    Private = AllocateZeroPool (sizeof (<<DRIVER_NAME>>_PRIVATE_DATA));
    if (Private == NULL) {
        return EFI_OUT_OF_RESOURCES;
    }
}

```

Example 116—Library allocation of private context data structure

The following example shows a code fragment from the `DiskIoDxe` driver in the `MdeModulePkg` that allocates and initializes the private context data structure from a template structure. A template structure is an instance of the private context structure

with most of the fields pre-initialized. This style produces UEFI Drivers that execute faster and produce smaller executables than UEFI Drivers that initialize each field of the private context data structure in the `start()` function.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/DevicePath.h>
#include <Protocol/DiskIo.h>
#include <Library/MemoryAllocationLib.h>

//
// Template for DiskIo private data structure.
// The pointer to BlockIo protocol interface is assigned dynamically.
//
DISK_IO_PRIVATE_DATA gDiskIoPrivateDataTemplate = {
    DISK_IO_PRIVATE_DATA_SIGNATURE,
    {
        EFI_DISK_IO_PROTOCOL_REVISION,
        DiskIoReadDisk,
        DiskIoWriteDisk
    },
    NULL
};

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 IO device instance.
    //
    Private = AllocateCopyPool (
        sizeof (DISK_IO_PRIVATE_DATA),
        &gDiskIoPrivateDataTemplate
    );
    if (Private == NULL) {
        Status = EFI_OUT_OF_RESOURCES;
        goto ErrorExit;
    }
}
```

Example 117—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 Driver Binding Protocol. The service typically used to free the private context data structures is `FreePool()` from the EDK II library `MemoryAllocationLib`.

Shown below is a generic template for freeing a private context data structure in the `Stop()` function of the Driver Binding Protocol. This code example shows only a fragment from the `Stop()` service. [Chapter 9](#) covers the services that are produced by the Driver Binding Protocol in more detail.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>

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,
        (VOID **)<<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
    //
    FreePool (Private);
}
```

```
    return Status;  
}
```

Example 118—Free a private context data structure

The following example shows a code fragment from the `DiskIoDxe` driver in the `MdeModulePkg` that frees the private context data structure.

```
#include <Uefi.h>  
#include <Protocol/DriverBinding.h>  
#include <Protocol/DiskIo.h>  
#include <Library/UefiBootServicesTableLib.h>  
#include <Library/MemoryAllocationLib.h>  
  
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,  
        (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);  
  
    FreePool (Private);  
  
    return Status;  
}
```

Example 119—Disk I/O free of a private context data structure

8.5 Retrieving private context data structures

The protocol functions produced by a UEFI driver need to access the private context data structure. These functions typically use the set of consumed protocols and the private data fields to perform the protocol function's required operation.

[Appendix A](#) contains a template for a `<>ProtocolName>.c` file 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 following example shows a code fragment from the `ReadDisk()` service of the `EFI_DISK_IO_PROTOCOL` that is produced by the `DiskIoDxe` driver in the `MdeModulePkg`. It uses the `CR()` based macro called `DISK_IO_PRIVATE_DATA_FROM_THIS()` and the `This` pointer to the `EFI_DISK_IO_PROTOCOL` to retrieve the `DISK_IO_PRIVATE_DATA` private context data structure.

```
#include <Uefi.h>
#include <Protocol/DiskIo.h>

EFI_STATUS
EFIAPI
DiskIoReadDisk (
    IN EFI_DISK_IO_PROTOCOL *This,
    IN UINT32               MediaId,
    IN UINT64               Offset,
    IN UINTN                BufferSize,
    OUT VOID                *Buffer
)
{
    DISK_IO_PRIVATE_DATA *Private;

    Private = DISK_IO_PRIVATE_DATA_FROM_THIS (This);
}
```

Example 120—Retrieving the Disk I/O private context data structure

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.

The example below shows a code fragment from the Driver Binding Protocol `Stop()` service of the `DiskIoDxe` driver in the `MdeModulePkg`. It uses the `CR()` based macro called `DISK_IO_PRIVATE_DATA_FROM_THIS()` and `EFI_DISK_IO_PROTOCOL` retrieved from `ControllerHandle` using the UEFI Boot Service `OpenProtocol()` to retrieve the `DISK_IO_PRIVATE_DATA` private context data structure.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/DiskIo.h>
#include <Library/UefiBootServicesTableLib.h>
```

```

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,
        (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);
}

```

Example 121—Retrieving the disk I/O private context data structure in Stop()

Driver Binding Protocol

The Driver Binding Protocol provides services to do the following:

- Connect a driver to a controller.
- Disconnect a driver from a controller.

UEFI drivers following 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. It is the one protocol used by the UEFI boot services `ConnectController()` and `DisconnectController()`. These UEFI boot services are used by the UEFI boot manager to connect the console and boot devices required to boot an operating system. The implementation of the Driver Binding Protocol varies depending upon the driver's category. [Chapter 6](#) of this guide describes the various driver categories.

9.1 Driver Binding Protocol Implementations

The implementation of the Driver Binding Protocol for a specific driver is typically found in the file `<<DriverName>>.c`. [Appendix A](#) contains a template for a `<<DriverName>>.c` file for a UEFI Driver. This file typically performs and contains the following:

- Adds a global variable for the `EFI_DRIVER_BINDING_PROTOCOL` instance to `<<DriverName>>.c`.
- An implementation of the `Supported()` service
- An implementation of the `Start()` service
- An implementation of the `Stop()` service
- Installs all the Driver Binding Protocols in the driver entry point
- If the UEFI Driver supports the unload feature, it then uninstalls all the Driver Binding Protocols in the `Unload()` function.

The example below shows the protocol interface structure for the Driver Binding Protocol for reference. It is composed of the three services called `Supported()`, `Start()`, and `Stop()`, along with the three data fields called `Version`, `ImageHandle`, and `DriverBindingHandle`.

```

typedef struct _EFI_DRIVER_BINDING_PROTOCOL  EFI_DRIVER_BINDING_PROTOCOL;

///
/// This protocol provides the services required to determine if a driver
/// supports a given controller. If a controller is supported, then it
/// also provides routines to start and stop the controller.
///
struct _EFI_DRIVER_BINDING_PROTOCOL {
    EFI_DRIVER_BINDING_SUPPORTED  Supported;
    EFI_DRIVER_BINDING_START      Start;
    EFI_DRIVER_BINDING_STOP       Stop;

    ///
    /// The version number of the UEFI driver that produced the
    /// EFI_DRIVER_BINDING_PROTOCOL. This field is used by
    /// the EFI boot service ConnectController() to determine
    /// the order that driver's Supported() service will be used when
    /// a controller needs to be started. EFI Driver Binding Protocol
    /// instances with higher Version values will be used before ones
    /// with lower Version values. The Version values of 0x0-
    /// 0x0f and 0xffffffff0-0xffffffff are reserved for
    /// platform/OEM specific drivers. The Version values of 0x10-
    /// 0xfffffffef are reserved for IHV-developed drivers.
    ///
    UINT32                         Version;

    ///
    /// The image handle of the UEFI driver that produced this instance
    /// of the EFI_DRIVER_BINDING_PROTOCOL.
    ///
    EFI_HANDLE                      ImageHandle;

    ///
    /// The handle on which this instance of the
    /// EFI_DRIVER_BINDING_PROTOCOL is installed. In most
    /// cases, this is the same handle as ImageHandle. However, for
    /// UEFI drivers that produce more than one instance of the
    /// EFI_DRIVER_BINDING_PROTOCOL, this value may not be
    /// the same as ImageHandle.
    ///
    EFI_HANDLE                      DriverBindingHandle;
};


```

Example 122—Driver Binding Protocol

UEFI Drivers declare a global variables for the Driver Binding Protocol instances produced. The `ImageHandle` and `DriverBindingHandle` fields are pre-initialized to `NULL`. A UEFI Driver can initialize the `ImageHandle` and `DriverBindingHandle` fields in the driver entry point, or use the EDK II library `UefiLib` functions to help initialize UEFI Drivers that fill and initialize the `ImageHandle` and `DriverBindingHandle` fields automatically. The `Version` field must be initialized by the UEFI Driver. Higher `Version` values signify a newer driver. This field is a 32-bit value, but the values 0x0–0x0F and 0xFFFFFFFF0–0xFFFFFFFF are reserved for UEFI drivers written by OEMs. IHVs may use the values

0x10–0xFFFFFFF. Each time a new version of a UEFI driver is released, the *Version* field must be increased. The following example shows how a Driver Binding Protocol is typically declared in a driver.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>

EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
    AbcSupported, // Supported()
    AbcStart,     // Start()
    AbcStop,      // Stop()
    0x10,         // Version
    NULL,         // ImageHandle
    NULL          // DriverBindingHandle
};
```

Example 123—Driver Binding Protocol declaration

The implementations of the Driver Binding Protocol change in complexity depending on the driver type. A device driver is the simplest to implement. A bus driver or a hybrid driver may be more complex because it has to manage both the bus controller and child controllers.

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 must be installed onto new handles.

The Driver Binding Protocol can be installed directly using the UEFI Boot Service **InstallMultipleProtocolInterfaces()**. However, the EDK II library **UefiLib** also provides a number of helper functions to install the Driver Binding Protocol and the optional UEFI Driver Model related protocols. The following helper functions are covered in more detail in [Chapter 7](#):

- **EfiLibInstallDriverBinding()**
- **EfiLibInstallAllDriverProtocols()**
- **EfiLibInstallDriverBindingComponentName2()**
- **EfiLibInstallAllDriverProtocols2()**

If an error is generated when installing any of the Driver Binding Protocol instances, the entire driver should fail and return a error status such as **EFI_ABORTED**. If a UEFI Driver implements the **Unload()** feature, any Driver Binding Protocol instances installed in the driver entry point must be uninstalled in the **Unload()** function.

9.2

Driver Binding Protocol 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. [Appendix A](#) contains the template for a UEFI Driver and includes the declaration of the Driver Binding Protocol instance, the Driver Binding Protocol services and the driver entry point that uses the EDK II library **UefiLib**.

functions to install the Driver Binding Protocol into the handle database and complete the initialization of the Driver Binding Protocol data fields.

The `Supported()`, `Start()`, and `Stop()` services are covered in detail in the EFI Driver Binding Protocol section of the *UEFI Specification*. Also included are code examples and the detailed algorithms to implement these services for device drivers and bus drivers

If a UEFI Driver produces multiple instances of the Driver Binding Protocol, they are all 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.

The `Supported()` service performs a quick check to see if a driver supports a controller. The `Supported()` service **must not** modify the state of the controller because the controller may already be managed by a different driver. If the `Supported()` service passes, the `Start()` service is called to ask the driver to bind to a specific controller. The `Stop()` service does the opposite of `Start()`. It disconnects a driver from a controller and frees any resources allocated in the `Start()` services.

TIP: Although the thought of initializing something as soon as it is supported in the `Supported()` service of the driver seems to make sense, the `Supported()` service is intended only to be a quick check to find out if a driver can make a connection to the specified controller, find out if it has already been called (started and in use), or if it is in use exclusively by another component. The `Supported()` service must return an error if the controller is already in use or is in use exclusively by another component.

Initializing or modifying tasks should only be done in the `Start()` service of the driver, not in the `Supported()` service.

TIP: This guide provides additional recommendations for implementing the Driver Binding Protocol for devices on industry standard busses such as PCI, USB, SCSI, and SATA. Please see the chapter on the specific bus type for additional details.

None of the Driver Binding Protocol services are allowed to use the console I/O protocols. A UEFI Driver may use the `DEBUG()` and `ASSERT()` macros from the EDK II library `DebugLib` to send messages to the standard error console if it is active. These macros are usually enabled during UEFI Driver development and are disabled when a UEFI Driver is released.

9.3

Testing Driver Binding Protocol

Once a Driver Binding Protocol is implemented, it can be tested using UEFI Shell commands. Use the UEFI Shell to load a UEFI Driver into memory and verify that the Driver Binding Protocol has been installed into the Handle Database correctly.

The UEFI Shell also provides commands to connect a driver to a device exercising the `Supported()` and `Start()` services, disconnect a driver from a device that exercises the `Stop()` service, and reconnect a driver to a device that exercises all the Driver Binding Protocol services. The details on each UEFI Shell command that may be used to test UEFI Drivers can be found in [Chapter 31](#) of this guide.

Full testing of a UEFI Driver is performed by booting UEFI operating systems and running the UEFI Self Certification Tests.

10 UEFI Service Binding Protocol

The Service Binding Protocol is not associated with a single GUID value. Instead, each Service Binding Protocol GUID value is paired with another protocol providing a specific set of services. The protocol interfaces for all Service Binding Protocols are identical and contain the services `CreateChild()` and `DestroyChild()`. When `CreateChild()` is called, a new handle is created with the associated protocol installed. When `DestroyChild()` is called, the associated protocol is uninstalled and the handle is freed. The *UEFI Specification* defines the following Service Binding Protocol GUIDs

Table 22—Service Binding Protocols

Service Binding Protocol	Associated Protocol
<code>EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_MANAGED_NETWORK_PROTOCOL</code>
<code>EFI_ARP_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_ARP_PROTOCOL</code>
<code>EFI_EAP_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_EAP_PROTOCOL</code>
<code>EFI_IP4_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_IP4_PROTOCOL</code>
<code>EFI_IP6_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_IP6_PROTOCOL</code>
<code>EFI_TCP4_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_TCP4_PROTOCOL</code>
<code>EFI_TCP6_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_TCP6_PROTOCOL</code>
<code>EFI_UDP4_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_UDP4_PROTOCOL</code>
<code>EFI_UDP6_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_UDP6_PROTOCOL</code>
<code>EFI_MTFTP4_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_MTFTP4_PROTOCOL</code>
<code>EFI_MTFTP6_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_MTFTP6_PROTOCOL</code>
<code>EFI_DHCP4_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_DHCP4_PROTOCOL</code>
<code>EFI_DHCP6_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_DHCP6_PROTOCOL</code>
<code>EFI_HASH_SERVICE_BINDING_PROTOCOL</code>	<code>EFI_HASH_PROTOCOL</code>

The Service Binding Protocol feature is required only if the associated protocol requires a Service Binding Protocol to produce its services and it defines a GUID value for that Service Binding Protocol. The table above lists the protocols defined in the *UEFI Specification* requiring the Service Binding Protocol feature. None of the other protocols defined by the *UEFI Specification* require a Service Binding Protocol.

For new protocols, a decision must be made to determine if the new protocol requires a Service Binding Protocol. The Driver Binding Protocol is usually sufficient for managing devices on common bus topologies and for the simple layering of protocols on a single device. When more complex tree or graph topologies are required and, with the expectation that services of the new protocol be required by multiple consumers, a Service Binding Protocol should be considered.

10.1 Service Binding Protocol Implementations

The implementation of the Service Binding Protocol for a specific driver is typically found in the file `<>DriverName>.c`. This file typically contains the following:

- Add global variable for the `EFI_SERVICE_BINDING_PROTOCOL` instance to `<>DriverName>.c`.
- Implementation of the `CreateChild()` service.
- Implementation of the `DestroyChild()` service.
- If the UEFI Driver follows the UEFI Driver Model, install all the Service Binding Protocol in the Driver Binding Protocol `Start()` function.
- If the UEFI Driver follows the UEFI Driver Model, uninstall all the Service Binding Protocol in the Driver Binding Protocol `Stop()` function.
- If the UEFI Driver is a Service Driver, install all the Service Binding Protocol in the driver entry point.
- If the UEFI Driver is a Service Driver that supports the unload feature, then uninstall all the Service Binding Protocol in the `Unload()` function.

The example below shows the protocol interface structure for the Service Binding Protocol for reference. It is composed of the two services called `CreateChild()` and `DestroyChild()`.

```
typedef struct _EFI_SERVICE_BINDING_PROTOCOL
  EFI_SERVICE_BINDING_PROTOCOL;

///
/// The EFI_SERVICE_BINDING_PROTOCOL provides member functions to create
/// and destroy child handles. A driver is responsible for adding
/// protocols to the child handle in CreateChild() and removing protocols
/// in DestroyChild(). It is also required that the CreateChild()
/// function opens the parent protocol BY_CHILD_CONTROLLER to establish
/// the parent-child relationship, and closes the protocol in
/// DestroyChild(). The pseudo code for CreateChild() and DestroyChild()
/// is provided to specify the required behavior, not to specify the
/// required implementation. Each consumer of a software protocol is
/// responsible for calling CreateChild() when it requires the protocol
/// and calling DestroyChild() when it is finished with that protocol.
///
struct _EFI_SERVICE_BINDING_PROTOCOL {
  EFI_SERVICE_BINDING_CREATE_CHILD          CreateChild;
  EFI_SERVICE_BINDING_DESTROY_CHILD         DestroyChild;
};
```

Example 124—Service Binding Protocol

10.2 Service Driver

If the UEFI Driver is a Service Driver, the Service Binding Protocol is installed in the driver entry point. The [following example](#) shows an implementation of a Service Binding Protocol that is installed into the Handle Database in the driver entry point. A

Service Binding Protocol is always paired with another protocol so, for this example, the paired protocol is the **ABC_PROTOCOL**.

Global variables are declared for the handle on which the Service Binding Protocol is installed, the instance of the Service Binding Protocol, and an instance of the **ABC_PROTOCOL**. The **ABC_PROTOCOL** instance is installed onto a new handle every time the Service Binding Protocol service **CreateChild()** is called. The **ABC_PROTOCOL** is uninstalled from a child handle every time the Service Binding Protocol service **DestroyChild()** is called.

```
#include <Uefi.h>
#include <Protocol/ServiceBinding.h>
#include <Library/UefiBootServicesTableLib.h>

typedef struct {
    UINT32 AbcField;
} ABC_PROTOCOL;

EFI_HANDLE gAbcServiceBindingHandle = NULL;

EFI_SERVICE_BINDING_PROTOCOL gAbcServiceBinding = {
    AbcCreateChild,
    AbcDestroyChild
};

ABC_PROTOCOL gAbc = {
    0
};

EFI_STATUS
EFIAPI
AbcCreateChild (
    IN     EFI_SERVICE_BINDING_PROTOCOL *This,
    IN OUT EFI_HANDLE                 *ChildHandle
)
{
    EFI_HANDLE NewHandle;

    NewHandle = NULL;
    return gBS->InstallMultipleProtocolInterfaces (
        &NewHandle,
        &gAbcProtocolGuid, &gAbc,
        NULL
    );
}

EFI_STATUS
EFIAPI
AbcDestroyChild (
    IN     EFI_SERVICE_BINDING_PROTOCOL *This,
    IN     EFI_HANDLE                  ChildHandle
)
{
    return gBS->UninstallMultipleProtocolInterfaces (
        ChildHandle,
        &gAbcProtocolGuid, &gAbc,
        NULL
    );
}
```

```

}

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
)
{
    //
    // Install Service Binding Protocol for ABC onto a new handle
    //
    return gBS->InstallMultipleProtocolInterfaces (
        &gAbcServiceBindingHandle,
        &gAbcServiceBindingProtocolGuid,
        &gAbcServiceBinding,
        NULL
    );
}

```

Example 125—Service Binding Protocol for Service Driver

10.3 UEFI Driver Model Driver

If the UEFI Driver follows the UEFI Driver Model, the Service Binding Protocol is installed in the Driver Binding Protocol `start()` function and uninstalled in the Driver Binding Protocol `stop()` function. This use case is covered in detail in the Service Binding Protocol section of the *UEFI Specification* and includes pseudo-code for implementations of the `CreateChild()` and `DestroyChild()` services. The EDK II also provides the following complete implementations of the Service Binding Protocol in drivers that follow the UEFI Driver Model:

- `MdeModulePkg\Universal\Network\MnpDxe`
- `MdeModulePkg\Universal\Network\ArpDxe`
- `MdeModulePkg\Universal\Network\Ip4Dxe`
- `NetworkPkg\Ip6Dxe`
- `MdeModulePkg\Universal\Network\Tcp4Dxe`
- `NetworkPkg\TcpDxe`
- `MdeModulePkg\Universal\Network\Udp4Dxe`
- `NetworkPkg\Udp6Dxe`
- `MdeModulePkg\Universal\Network\Mtftp4Dxe`
- `NetworkPkg\Mtftp6Dxe`
- `MdeModulePkg\Universal\Network\DHcp4Dxe`
- `NetworkPkg\DHcp6Dxe`

UEFI Driver and Controller Names

Both Component Name Protocols are optional features that allow UEFI Drivers following the UEFI Driver Model to provide a localized Unicode name string for the UEFI Driver and the devices the UEFI Driver manages. Use of these protocols depends on the UEFI Driver Model concepts. Service Drivers, Root Bridge Drivers, and Initializing Drivers never produce the Component Name Protocols. Implementation of this optional feature is **recommended** for all UEFI Drivers that follow the UEFI Driver Model.

Note: *Human-readable names should be limited to about 40 Unicode characters in length. This makes it easier for consumers of this protocol to display these names on standard console devices.*

The Component Name Protocol and the Component Name 2 Protocol are very similar. The only difference is the format of language code passed into the protocol services to request the name of a UEFI Driver or the name of a device that a UEFI Driver manages. The use of a language code allows the implementation of the Component Name Protocols to provide names of drivers and devices in many different languages.

The Component Name Protocol uses ISO 639-2 language codes (i.e. `eng`, `fra`). The Component Name 2 Protocol uses RFC 4646 language codes (i.e. `en`, `en-US`, `fr`). If names are provided for platforms conforming to the *EFI 1.10 Specification*, the Component Name s Protocol is required. If names are provided for platforms that conforming to the *UEFI 2.0 Specification* or above, the Component Name 2 Protocol is required. Since the only difference is the language code for the names, UEFI Drivers required to provide names typically produce both protocols and the both use the same underlying functions and Unicode name strings.

The Component Name Protocols are installed onto handles in the driver entry point of a UEFI Driver. [Chapter 7](#) describes details on the EDK II library `UefiLib` that provides helper functions to initialize UEFI Drivers following the UEFI Driver Model including installation of Component Name Protocols.

Component Name Protocols may be used by a UEFI Boot Manager to display human readable names for drivers and devices in a specific language. A platform vendor may also take advantage of Component Name Protocols from UEFI Applications, such as system utilities or diagnostics, when human readable names of UEFI drivers or devices are required.

The UEFI Shell provides several commands that use the Component Name Protocols. For example, the `drivers` command displays the inventory of UEFI drivers in a platform and uses the Component Name Protocols to display the name of a UEFI Driver if the UEFI Driver produced the Component Name Protocols. Likewise, the UEFI Shell command `devices` displays the inventory of devices in a platform and uses the Component Name Protocols to display the name of the devices if a UEFI Driver managing the device produced the Component Name Protocols.

If a controller is managed by more than one UEFI Driver, there may be multiple instances of the Component Name Protocols that apply to a single controller. The consumers of the Component Name Protocols have to decide how the multiple drivers providing names are presented to the user. For example, a PCI bus driver may produce a name for a PCI slot such as "PCI Slot #2," and the driver for a 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.

[Appendix B](#) contains a table of example drivers from the EDK II along with the features that each implement. The EDK II provides example drivers with full implementations of the Component Name Protocols.

11.1 Component Name Protocol Implementations

The implementation of the Component Name Protocols for a specific driver is typically found in the file `ComponentName.c`. [Appendix A](#) contains a template for a `ComponentName.c` file for a UEFI Driver. This file typically contains the following:

- Add global variable for the `EFI_COMPONENT_NAME_PROTOCOL` instance to `ComponentName.c`.
- Add global variable for the `EFI_COMPONENT_NAME2_PROTOCOL` instance to `ComponentName.c`.
- `EFI_COMPONENT_NAME2_PROTOCOL` instance
- Add static table of UEFI Driver names as Unicode strings to `ComponentName.c`.
- Add static table of controller names as Unicode strings to `ComponentName.c`.
- Implementation of the `GetDriverName()` service
- Implementation of the `GetControllerName()` service
- Install all the Component Name Protocols in the driver entry point.
- If the UEFI Driver supports the unload feature, uninstall all the Component Name Protocols in the `Unload()` function.

The Component Name Protocols provide names in one or more languages. At a minimum, the protocols should support the English language. The Component Name Protocols advertise the languages they supports in a data field called `SupportedLanguages`. This data filed is a null-terminated ASCII string that contains one or more 3 character ISO 639-2 language codes with no separator character. The Component Name 2 Protocol also 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 RFC 4646 language codes separated by semicolons (';').

A consumer of the Component Name Protocols may parse the `SupportedLanguages` data field to determine 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 Protocols to see if names are available in the requested language.

For reference, [Example 126](#), below, shows the protocol interface structure for the Component Name Protocol and [Example 127](#) shows the protocol interface structure for the Component Name 2 Protocol. Both are composed of the two services called `GetDriverName()` and `GetControllerName()` and a data field called `SupportedLanguages`.

```

typedef struct _EFI_COMPONENT_NAME_PROTOCOL  EFI_COMPONENT_NAME_PROTOCOL;

///
/// This protocol is used to retrieve user readable names of drivers
/// and controllers managed by UEFI Drivers.
///
struct _EFI_COMPONENT_NAME_PROTOCOL {
    EFI_COMPONENT_NAME_GET_DRIVER_NAME      GetDriverName;
    EFI_COMPONENT_NAME_GET_CONTROLLER_NAME GetControllerName;
    ///
    /// A Null-terminated ASCII string that contains one or more
    /// ISO 639-2 language codes. This is the list of language codes
    /// that this protocol supports.
    ///
    CHAR8                                *SupportedLanguages;
};

```

Example 126—Component Name Protocol

```

typedef struct _EFI_COMPONENT_NAME2_PROTOCOL
EFI_COMPONENT_NAME2_PROTOCOL;

///
/// This protocol is used to retrieve user readable names of drivers
/// and controllers managed by UEFI Drivers.
///
struct _EFI_COMPONENT_NAME2_PROTOCOL {
    EFI_COMPONENT_NAME2_GET_DRIVER_NAME      GetDriverName;
    EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME GetControllerName;
    ///
    /// A Null-terminated ASCII string array that contains one or more
    /// supported language codes. This is the list of language codes that
    /// this protocol supports. The number of languages supported by a
    /// driver is up to the driver writer. SupportedLanguages is
    /// specified in RFC 4646 format.
    ///
    CHAR8                                *SupportedLanguages;
};

```

Example 127—Component Name 2 Protocol

UEFI Drivers declare global variables for the Component Name Protocol and Component Name 2 Protocol instances that are produced. The *SupportedLanguages* fields are typically initialized by the UEFI Driver in the declaration for the specific set of languages the UEFI Driver supports. The following [following example](#) shows how the Component Name Protocols are typically declared in a driver and, in this case, declared to support both English and French.

```

#include <Uefi.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/ComponentName.h>

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL gAbcComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME) AbcGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) AbcGetControllerName,
    "engfra"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL gAbcComponentName2 = {
    AbcGetDriverName,
    AbcGetControllerName,
    "en;fr"
};

```

Example 128—Driver Diagnostics Protocol declaration

The implementations of the Component Name Protocols change in complexity depending on the type of UEFI Driver Model driver and the specific Component Name Protocol features implemented. A device driver is the simplest to implement. A bus driver or a hybrid driver may be more complex because it may provide names for both the bus controller and the child controllers. These implementations are discussed later in this section.

The `EFI_COMPONENT_NAME_PROTOCOL` and `EFI_COMPONENT_NAME2_PROTOCOL` are installed onto the driver's image handle. It is possible for a driver to produce more than one instance of the Component Name Protocols. All additional instances of the Component Name Protocols must be installed onto new handles.

The Component Name Protocols can be installed directly using the UEFI Boot Service `InstallMultipleProtocolInterfaces()`. However, the EDK II library `UefiLib` provides a number of helper functions to install the Component Name Protocols. The helper functions covered in more detail in [Chapter 7](#) are:

- `EfiLibInstallDriverBinding()`
- `EfiLibInstallAllDriverProtocols()`
- `EfiLibInstallDriverBindingComponentName2()`
- `EfiLibInstallAllDriverProtocols2()`

If an error is generated installing any of the Component Name Protocol instances the entire driver should fail and return an error status such as `EFI_ABORTED`. If a UEFI Driver implements the `Unload()` feature, any Component Name Protocol instances installed in the driver entry point must be uninstalled in the `Unload()` function.

The simplest implementation of the Component Name Protocols provides the name of the UEFI Driver. The next most complex implementation is that for a device driver providing both the name of the UEFI Driver and the names of the controllers under UEFI Driver management. The most complex implementation is that of a bus or a hybrid driver producing names for the UEFI Driver, names for the bus controllers it is managing, and names for the child controllers the driver has produced. All three of these implementations are discussed in the sections that follow.

The EDK II library **UefiLib** provides functions to simplify the implementation of the Component Name Protocols. These library functions provide services to register Unicode strings in a table, lookup Unicode strings in a table, and free tables of Unicode strings. Some UEFI Drivers have fixed names for the UEFI Driver itself and the controllers that they manage. Other UEFI Drivers may dynamically create names based on information retrieved from the platform or the controller itself. The EDK II library **UefiLib** functions managing tables of Unicode strings are:

- **LookupUnicodeString()**
- **LookupUnicodeString2()**
- **AddUnicodeString()**
- **AddUnicodeString2()**
- **FreeUnicodeStringTable()**

UEFI Drivers producing dynamic names for controllers or children register those dynamic names in the Driver Binding Protocol **Start()** function and are freed in the Driver Binding **Stop()** function. In addition, dynamic name tables require extra fields in the driver's private context data structure pointing to the dynamic name tables. See [Chapter 8](#) of this guide for details on the design of private context data structures.

11.2

GetDriverName() Implementations

The **GetDriverName()** service retrieves the name of a UEFI Driver. It may be used to retrieve the name of a UEFI Driver even if the UEFI Driver is not managing any devices. Example 129, below, shows a typical implementation of the **GetDriverName()** service for the Component Name 2 Protocol along with a table of Unicode strings for the UEFI Driver name in English, French, and Spanish. The recommended implementation style shown here allows the same **GetDriverName()** service implementation to be shared between the Component Name Protocol and the Component Name 2 Protocol. The **UefiLib** function **LookupUnicodeString2()** supports looking up strings using either ISO 639-2 or RFC 4646 language code formats.

The static table of driver names contains two elements per entry. The first is an ASCII string containing one or more language codes separated by ';' characters. The language codes may be in the ISO639-2 or the RFC 4646 format.

The second element is a Unicode string representing the name of the UEFI Driver for the set of languages specified by the first element. The static table is terminated by two **NULL** elements. The format is very size efficient because each Unicode string name for the UEFI Driver can be associated with many language codes.

```
#include <Uefi.h>
#include <Protocol/ComponentName2.h>
#include <Library/UefiLib.h>

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_UNICODE_STRING_TABLE mAbcDriverNameTable[] = {
    { "eng;en", (CHAR16 *)L"ABC Driver in English" },
    { "fra;fr", (CHAR16 *)L"ABC Driver in French" },
    { "spa;sp", (CHAR16 *)L"ABC Driver in Spanish" },
    { NULL, NULL }
};
```

```

EFI_STATUS
EFIAPI
AbcGetDriverName (
    IN  EFI_COMPONENT_NAME2_PROTOCOL   *This,
    IN  CHAR8                         *Language,
    OUT CHAR16                        **DriverName
)
{
    return LookupUnicodeString2 (
        Language,
        This->SupportedLanguages,
        mAbcDriverNameTable,
        DriverName,
        (BOOLEAN)(This != &gAbcComponentName2)
    );
}

```

Example 129—GetDriverName() for Device, Bus, or Hybrid Driver

11.3 GetControllerName() Implementations

The `GetControllerName()` service retrieves the name of a controller a driver is managing or a child the driver has produced. The example below shows an empty implementation of the `GetControllerName()` service for the Component Name 2 Protocol. The recommended implementation style shown here allows the same `GetControllerName()` service implementation to be shared between both the Component Name Protocol and the Component Name 2 Protocol.

```

#include <Uefi.h>
#include <Protocol/ComponentName2.h>

EFI_STATUS
EFIAPI
AbcGetControllerName (
    IN  EFI_COMPONENT_NAME2_PROTOCOL   *This,
    IN  EFI_HANDLE                   ControllerHandle,
    IN  EFI_HANDLE                   ChildHandle      OPTIONAL,
    IN  CHAR8                        *Language,
    OUT CHAR16                       **ControllerName
)
{
}

```

Example 130—GetControllerName () Service

The Component Name Protocols are available only for devices currently under a driver's management. Because UEFI supports connecting the minimum number of drivers and devices required to establish console and gain access to the boot device, there may be many unconnected devices for which a name may not be retrieved.

11.3.1 Device Drivers

Device drivers implementing `GetControllerName()` must verify that `ChildHandle` is `NULL` and that `ControllerHandle` represents a device the device driver is currently managing. In addition, `GetControllerName()` must verify that the requested `Language` is in the set of languages the UEFI Driver supports. The example below shows the steps required to check these parameters. If the checks pass, the name of the controller is returned. In this specific example, the driver opens the PCI I/O Protocol in its Driver Binding `Start()` function. This is why `gEfiPciIoProtocolGuid` is used in the call to the EDK II Library `UefiLib` function `EfiTestManagedDevice()` that checks to see if the UEFI Drivers providing the `GetControllerName()` service is currently managing `ControllerHandle`. Just like the `GetDriverName()` example in the previous section, a static table of Unicode strings for the controller names is declared as a global variable and the `LookupUnicodeString2()` service is used to lookup the name of the controller in the requested `Language`.

```
#include <Uefi.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/PciIo.h>
#include <Library/UefiLib.h>

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_UNICODE_STRING_TABLE mAbcControllerNameTable[] = {
    { "eng;en", (CHAR16 *)L"ABC Controller in English"}, 
    { "fra;fr", (CHAR16 *)L"ABC Controller in French"}, 
    { "spa;sp", (CHAR16 *)L"ABC Controller in Spanish"}, 
    { NULL, NULL }
};

EFI_STATUS
EFIAPI
AbcGetControllerName (
    IN EFI_COMPONENT_NAME2_PROTOCOL *This,
    IN EFI_HANDLE                  ControllerHandle,
    IN EFI_HANDLE                  ChildHandle OPTIONAL,
    IN CHAR8                      *Language,
    OUT CHAR16                     **ControllerName
)
{
    EFI_STATUS Status;

    //
    // ChildHandle must be NULL for a Device Driver
    //
    if (ChildHandle != NULL) {
        return EFI_UNSUPPORTED;
    }

    //
    // Make sure this driver is currently managing ControllerHandle
    //
    Status = EfiTestManagedDevice (
        ControllerHandle,
        gAbcDriverBinding.DriverBindingHandle,
        &gEfiPciIoProtocolGuid
    );
    if (EFI_ERROR (Status)) {

```

```

        return Status;
    }

    return LookupUnicodeString2 (
        Language,
        This->SupportedLanguages,
        mAbcControllerNameTable,
        ControllerName,
        (BOOLEAN)(This != &gAbcComponentName2)
    );
}

```

Example 131—GetControllerName() for a Device Driver

If the private context structure is required, use the UEFI Boot Service `OpenProtocol()` to open one of the protocols on `ControllerHandle` produced by the UEFI Driver and then use a `CR()` based macro to retrieve a pointer to the private context structure.

Some device drivers can extract name information from the devices they manage and are then able to provide more specific device names. The dynamic generation of controller names does increase the complexity of the UEFI Driver implementation, but it may provide users with the detailed information they require to identify a specific device. For example, a driver for a mass storage device may be able to produce a static name such as "Hard Disk," but a more specific name, such as "XYZ Manufacturer SATA Model 123 Hard Disk", may be much more useful.

To support the dynamic generation of controller names, a few additional steps must be taken. First, a pointer to the dynamic table of names must be added to the private context data structure for the controllers a device driver manages. The example below shows the addition of an `EFI_UNICODE_STRING_TABLE` field to the private context data structure discussed in [Chapter 8](#) of this guide.

```

#define ABC_PRIVATE_DATA_SIGNATURE  SIGNATURE_32 ('A','B','C',' ')
#define ABC_PRIVATE_DATA_FROM_PCI_IO_THIS(a) \
    CR (a, ABC_PRIVATE_DATA, PciIo, ABC_PRIVATE_DATA_SIGNATURE)

```

Example 132—Controller names in private context data structure

The next update is to the `start()` service of the Driver Binding Protocol. It needs to add a controller name in each supported language to `ControllerNameTable` in the private context data structure. Use the `UefiLib` function `AddUnicodeString2()` to add one or more names to a table. The `ControllerNameTable` must be initialized to `NULL` before the first name is added.

The following example shows the addition of an English name to a dynamically allocated table of Unicode names. If more than one language is supported, then `AddUnicodeString2()` is called 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.

```
#include <Uefi.h>
#include <Library/UefiLib.h>

ABC_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 using ISO 639-2 language code
//
AddUnicodeString2 (
    "eng",
    gAbcComponentName.SupportedLanguages,
    &Private->ControllerNameTable,
    ControllerName,
    TRUE
);

//
// Add the device name to the table of names stored in the private
// context data structure using RFC 4646 language code
//
AddUnicodeString2 (
    "en",
    gAbcComponentName2.SupportedLanguages,
    &Private->ControllerNameTable,
    ControllerName,
    FALSE
);
```

Example 133—Adding a controller name to a dynamic controller name table

The `stop()` service of the 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 built in the `start()` service must be freed. Use the UEFI driver library function `FreeUnicodeStringTable()` to free the table of controller names.

The code to add to the Driver Binding Protocol `stop()` service follows. The private context data structure is required by the `stop()` service so the private context data structure can be freed. The call to `FreeUnicodeStringTable()` should be made just before the private context data structure is freed.

```

#include <Uefi.h>
#include <Library/UefiLib.h>

ABC_PRIVATE_DATA *Private

FreeUnicodeStringTable (Private->ControllerNameTable);

```

Example 134—Freeing a dynamic controller name table

Lastly, the `GetControllerName()` service is slightly different because the dynamic table of controller names from the private context structure is used instead of the static table of controller names. 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. The private context data structure can then be used with the `UefiLib` function `LookupUnicodeString2()` to look up the controller's name in the dynamic table of controller names.

The example below shows the `GetControllerName()` service that retrieves the controller name from a dynamic table stored in the private context data structure.

```

#include <Uefi.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/PciIo.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI
AbcGetControllerName (
    IN  EFI_COMPONENT_NAME2_PROTOCOL   *This,
    IN  EFI_HANDLE                   ControllerHandle,
    IN  EFI_HANDLE                   ChildHandle      OPTIONAL,
    IN  CHAR8                        *Language,
    OUT CHAR16                       **ControllerName
)
{
    EFI_STATUS          Status;
    EFI_PCI_IO_PROTOCOL *PciIo;
    ABC_PRIVATE_DATA   *Private;

    //
    // ChildHandle must be NULL for a Device Driver
    //
    if (ChildHandle != NULL) {
        return EFI_UNSUPPORTED;
    }

    //
    // Make sure this driver is currently managing ControllerHandle
    //
    Status = EfiTestManagedDevice (
        ControllerHandle,
        gAbcDriverBinding.DriverBindingHandle,
        &gEfiPciIoProtocolGuid
    );
}

```

```

if (EFI_ERROR (Status)) {
    return Status;
}

//
// Retrieve an instance of a produced protocol from ControllerHandle
//
Status = gBS->OpenProtocol (
    ControllerHandle,
    &gEfiPciIoProtocolGuid,
    (VOID **) &PciIo,
    gAbcDriverBinding.DriverBindingHandle,
    ControllerHandle,
    EFI_OPEN_PROTOCOL_GET_PROTOCOL
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Retrieve the private context data structure for ControllerHandle
//
Private = ABC_PRIVATE_DATA_FROM_PCI_IO_THIS (PciIo);

//
// Look up the controller name from a dynamic table of controller names
//
return LookupUnicodeString2 (
    Language,
    This->SupportedLanguages,
    Private->ControllerNameTable,
    ControllerName,
    (BOOLEAN) (This != &gAbcComponentName2)
);
}

```

Example 135—Device driver with dynamic controller names

11.3.2 Bus Drivers and Hybrid Drivers

There are many levels of support a bus driver or hybrid driver may provide for the Component Name Protocols. These drivers can choose to provide a driver name as described in the section of this chapter on `GetDriverName()`. They can also choose to provide names for the bus controllers they manage and to not provide any names for the children they produce (such as the device drivers described the previous section). This discussion explains what bus drivers and hybrid drivers need to do to provide human-readable names for the child handles they produce. The human-readable names for child handles can be provided through static or dynamic controller name tables.

Note: *It is recommended that bus drivers and hybrid drivers provide controller names for both the bus controller and the child controllers these types of drivers produce. Implementing controller names for only the bus controller or only the child controllers is discouraged.*

Bus drivers and hybrid drivers implementing the Component Name Protocols must verify that `ControllerHandle` and `ChildHandle` represent a device the driver is currently managing. In addition, `GetControllerName()` must verify the requested `Language` is in the set of languages the UEFI Driver supports. The following example shows the steps required to check these parameters. If these checks pass, the controller name is returned in the requested language. In this specific example, the driver opens the PCI I/O Protocol in its Driver Binding Start() function. This is why `gEfiPciIoProtocolGuid` is used in the call to the EDK II Library `UefiLib` function `EfiTestManagedDevice()` that checks to see if the UEFI Drivers providing the `GetControllerName()` service is currently managing `ControllerHandle`. If the private context structure is required, then typically the UEFI Boot Service `OpenProtocol()` is used to open one of the protocols on `ControllerHandle` that the UEFI Driver produced and then uses a `CR()` based macro to retrieve a pointer to the private context structure.

Note: *If ChildHandle is NULL, a request is made for the name of the bus controller. If ChildHandle is not NULL, a request is made for the name of a child controller managed by the UEFI Driver.*

```
#include <Uefi.h>
#include <Protocol/ComponentName2.h>
#include <Protocol/PciIo.h>
#include <Library/UefiLib.h>

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_UNICODE_STRING_TABLE mAbcControllerNameTable[] = {
{ "eng;en", (CHAR16 *)L"ABC Bus Controller in English"},
{ "fra;fr", (CHAR16 *)L"ABC Bus Controller in French"},
{ "spa;sp", (CHAR16 *)L"ABC Bus Controller in Spanish"},
{ NULL, NULL }
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_UNICODE_STRING_TABLE mAbcChildNameTable[] = {
{ "eng;en", (CHAR16 *)L"ABC Child Controller in English"},
{ "fra;fr", (CHAR16 *)L"ABC Child Controller in French"},
{ "spa;sp", (CHAR16 *)L"ABC Child Controller in Spanish"},
{ NULL, NULL }
};

EFI_STATUS
EFIAPI
AbcGetControllerName (
    IN  EFI_COMPONENT_NAME2_PROTOCOL   *This,
    IN  EFI_HANDLE                   ControllerHandle,
    IN  EFI_HANDLE                   ChildHandle      OPTIONAL,
    IN  CHAR8                        *Language,
    OUT CHAR16                       **ControllerName
)
{
    EFI_STATUS             Status;
    EFI_UNICODE_STRING_TABLE *NameTable;

    //
    // Make sure this driver is currently managing ControllerHandle
    //
    Status = EfiTestManagedDevice (
        ControllerHandle,
        gAbcDriverBinding.DriverBindingHandle,
        &gEfiPciIoProtocolGuid
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }
}
```

```

if (ChildHandle == NULL) {
    NameTable = mAbcControllerNameTable;
} else {
    //
    // If ChildHandle is not NULL, then make sure this driver produced ChildHandle
    //
    Status = EfiTestChildHandle (
        ControllerHandle,
        ChildHandle,
        &gEfiPciIoProtocolGuid
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }
    NameTable = mAbcChildNameTable;
}

return LookupUnicodeString2 (
    Language,
    This->SupportedLanguages,
    NameTable,
    ControllerName,
    (BOOLEAN)(This != &gAbcComponentName2)
);
}

```

Example 136—GetControllerName() for a Bus Driver or Hybrid Driver

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 Driver Binding Protocol. The previous section explains how this update is done for the controller names. The exact same technique can be applied to child controllers.

11.4 Testing Component Name Protocols

Use the UEFI Shell's `drivers` and `devices` commands to exercise the Component Name Protocols. Running these commands with no options shows the sets of drivers and devices in the platform. The names are shown in the currently set platform language.

These commands also support a `-l` option to request names in an alternate language. [Figure 15](#), following, shows an example using the UEFI Shell command `drivers` on the EDK II Nt32 platform. [Figure 16](#) then shows an example of using the UEFI Shell command `devices` on the EDK II Nt32 platform. The details on each UEFI Shell command available to test UEFI Drivers can be found in [Chapter 31](#) of this guide.

	T	D				IMAGE NAME		
V	VERSION	E	G	G	#D	#C	DRIVER NAME	
3E	0000000A	D	-	-	3	-	Platform Console Management Driver	ConPlatformDxe
3F	0000000A	D	-	-	3	-	Platform Console Management Driver	ConPlatformDxe
40	0000000A	B	-	-	3	3	Console Splitter Driver	ConSplitterDxe
41	0000000A	? -	-	-	-	-	Console Splitter Driver	ConSplitterDxe
42	0000000A	? -	-	-	-	-	Console Splitter Driver	ConSplitterDxe
43	0000000A	B	-	-	3	3	Console Splitter Driver	ConSplitterDxe
44	0000000A	? -	-	-	-	-	Console Splitter Driver	ConSplitterDxe
47	0000000A	D	-	-	2	-	UGA Console Driver	GraphicsConsoleDxe
48	0000000A	B	-	-	1	1	Serial Terminal Driver	TerminalDxe
49	0000000A	D	-	-	2	-	Generic Disk I/O Driver	DiskIoHiiDxe
4A	0000000B	? -	-	-	-	-	Partition Driver (MBR/GPT/El Torito)	PartitionDxe
4D	0000000A	? -	-	-	-	-	PCI Bus Driver	PciBusDxe
4F	0000000A	? -	-	-	-	-	SCSI Bus Driver	ScsiBus
50	0000000A	? -	-	-	-	-	Scsi Disk Driver	ScsiDisk
51	0000000A	? - X	-	-	-	-	PCI IDE/ATAPI Bus Driver	IdeBusDxe
52	0000000A	B	-	-	1	12	Windows Bus Driver	WinNtBusDriverDxe
53	0000000A	D - X	2	-	-	-	Windows Block I/O Driver	WinNtBlockIo
54	0000000A	B -	-	-	1	1	Windows Serial I/O Driver	WinNtSerialIodxe
55	0000000A	D -	-	-	2	-	Windows GOP Driver	WinNtGopDxe

Figure 15—Testing Component Name Protocol GetDriverName()

C	T	D					
L	E	G	G	#P	#D	#C	Device Name
1C	R	-	-	1	12	-	VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)
45	D	-	-	3	-	-	Primary Console Input Device
46	D	-	-	3	-	-	Primary Console Output Device
6B	B	-	-	1	1	1	COM1
6C	B	-	-	1	1	1	COM1
6D	B	-	-	1	4	2	PC-ANSI Serial Console
6E	B	-	-	1	6	2	UGA Window 1
6F	B	-	-	1	6	2	UGA Window 2
70	D	-	-	1	-	-	CDM2
71	D	-	-	1	-	-	Bus Driver Console Window

Figure 16—Testing Component Name Protocol GetControllerName()

12

UEFI Driver Configuration

The configuration of UEFI Drivers is typically provided through HII. If a UEFI Driver requires interaction with a user to properly configure a device for use in the UEFI pre-boot environment, HII packages must be registered and the HII Config Access Protocol must be implemented. The requirement for HII packages and the HII Config Access Protocol applies to UEFI Drivers required to be compatible with platforms conformant with the *UEFI 2.1 Specification* or higher. This chapter focuses on guidelines for UEFI Drivers required to produce HII based configuration methods.

If a UEFI Driver is required to be compatible with platforms conformant with the *UEFI 2.0 Specification*, the Driver Configuration 2 Protocol must be implemented. If a UEFI Driver is required to be compatible with platforms conformant with the *UEFI 1.1 Specification*, the Driver Configuration Protocol must be implemented.

UEFI platform firmware supporting HII provides an HII forms browser. This component uses UEFI consoles to display configuration forms to the user and allows the user to navigate between forms and within forms to answer questions related to the configuration of devices.

12.1 HII overview

A UEFI driver is not allowed to directly invoke a platform's forms browser. Instead, a UEFI Driver provides sets of forms (the equivalent of Web pages) to HII. If and when the forms browser is run, the web pages are displayed and configuration takes place. The benefits of using forms instead of the Simple Text Input Protocol and Simple Text Output Protocols include that:

- The forms allow use of a pre-existing GUI—the system already has a browser. This means a UEFI Driver can take advantage of the browser's features and allows the system to have a more consistent look and feel for the user.
- The forms are device-neutral. The browser can manage the forms appropriately for any device—for instance, a smart phone versus a laptop.
- The forms allow for remote configuration of devices. Instead of requiring that the user to go the physical machine and press (for example) Ctrl-Alt-F4, the text input can be handled remotely via the browser.

HII is designed to enable support of the data structures required to support fully localized text and graphical user interfaces to the user. This consists of four types of support:

1. **Keyboard:** HII supports keyboard mappings—the keyboard reflects the language the user is expecting to use. For example, French and English mappings differ in the Q, A, and Z keys. Keyboards simply return the

location of the key, not its Unicode value. The HII support for key mapping allows translation from key location to Unicode value. There is no support for IMEs.

2. **Fonts:** HII supports fonts for the approximately 37,000 Unicode printable characters in Unicode UCS-2. The system carries the Latin-1 (Western European) character set. Other characters must be provided if they are to be displayed. HII also supports narrow and wide characters to support logographic languages (such as Chinese, Japanese, and Korean).
3. **Strings:** HII expects strings to be compressed Unicode stored by language. Drivers reference strings by IDs, which requires less storage. The actual string selected is defined by the ID and by the selected language.
4. **Forms:** HII defines its own forms language known as IFR. Although similar to web-based forms languages (such as HTML), IFR is stored in binary. IFR supports the usual tags, headers, and so on, found in a normal forms markup language. However, IFR also has special support for items common to configuration including multiple defaults and context-sensitive help. Unlike most forms languages, HII refers to strings via ID, so the same form can be used for multiple languages. HII also supports a rich set of operations for validating results. If all else fails, HII can reference callbacks into the submitting driver's code.

Note: *IFR is a variable-length encoding of HTML-like tags. While experts can write in this language (a bit like using DBs to write assembly language), most developers use a high level language known as VFR. VFR compiles into IFR and makes writing UEFI forms similar to writing HTML. The EDK II build tools provide full support for VFR along with a VFR to IFR compiler.*

HII data is stored in a central HII database dynamically created upon each reboot. HII protocols allow for a driver's HII data to be submitted, manipulated, and extracted.

Configuration in a UEFI system is the province of a single setup browser. Drivers submit their HII data to the HII protocols. The browser then parses through the forms in the same way an internet browser would parse web data. The setup browser communicates with the drivers to obtain current configuration information and to provide updates when the session completes.

12.1.1 HII Database and Package Lists

The HII database is built dynamically as the system boots. A UEFI Driver is required to register lists of HII packages into the HII Database. A package list is a list of packages providing different types of binary data. The data types supported include font, string, image, keyboard, and forms data.

Note: *The package could also contain some keyboard data but keyboard layouts are typically outside the scope of a driver (and typically up to the platform to determine). For example, keyboard data could represent the French keyboard, a simplified set of Hiragana and Katakana characters for a Japanese keyboard, and so on. The Unicode values of keyboard data are mapped to the characters of each supported language and displayed to the screen.*

The goal of the package is to create a single form with multiple sets of strings. For example, the goal for fonts is to create a single form with multiple sets of strings, each set for a different supported language. The sets of strings are published to the HII database, which also contains the strings, fonts, and characters from other drivers. The setup browser can then access the HII Database to display the forms in the appropriate language and font.

In general, data in a package is not modified after it is registered. For example, data that probably won't change during configuration include the questions that are presented to a user, the layout of the forms, the font list, and so on.

12.2 General steps for implementing HII functionality

To include HII functionality in a driver, follow these general steps:

1. Decide if the user should be allowed to change configuration data.
 2. Identify a set of options that can be changed. These options are stored in NVRAM. The NVRAM storage may be either local to the device being configured (recommended), or it may be global to the platform such as UEFI Variables (not recommended). UEFI Variables must be used only to store configuration data for UEFI Drivers integrated into a platform.
 3. Define the C data structure for the configurable data. Drivers have flexibility on how they process configurable data. For example, data can be managed as name, value pairs or as a data structure. The full C data structure is defined in a .h file for the UEFI Driver usually in <>DriverName>>.h.
 4. Determine the order of the questions presented to the user—for example, the order in which to present the fields and their values. For example, select on or off, yes or no, or enter a specific value, and so on. This information is typically stored in the file called <>DriverName>>.Vfr. The order in which the information is listed in the <>DriverName>>.Vfr file is the order in which each configurable field is displayed in the form.
- TIP:** When designing questions, remember that the way the user sees the data may vary considerably depending on the device used. For example, it could vary from a few lines on a plasma display on the front panel of a home electronics device to a full, rich GUI interface on a remote console.
5. Define the strings for the form including the title of the form, help information for the form title; the titles for each configurable field and the help information (if any) for each configurable field. This information is typically stored in the <>DriverName>>.Uni. The example below shows a portion of the Unicode string file from a sample driver in the MdeModulePkg on the path MdeModulePkg/Universal/DriverSampleDxe.

```
#langdef en-US "English"
#langdef fr-FR "Francais"
$string STR_FORM_SET_TITLE      #language en-US "Browser Testcase Engine"
                                #language fr-FR "Browser Testcase Engine"
$string STR_FORM_SET_TITLE_HELP #language en-US "This is a sample driver which is
                                used to test the browser op-code operations. This is for development purposes and
                                not to be distributed in any form other than a test application. Here is a set of
                                wide \wideAAAAAAA\narrow and narrow AAA!"
```

```

#language fr-FR "This is a sample driver which is
used to test the browser op-code operations. This is for development purposes and
not to be distributed in any form other than a test application. Here is a set of
wide \wideAAAAAAA\wide and narrow AAA!"#string STR_FORM1_TITLE
#language en-US "My First Setup Page"
#language fr-FR "Mi Primero Arreglo Página"
#language en-US "My Second Setup Page"
#language fr-FR "Mi Segunda Paginación la
Disposición;
#string STR_FORM2_TITLE
#language en-US "My Third Setup Page"
#language fr-FR "Mi Tercera Paginación la
Disposición;
#string STR_DYNAMIC_TITLE
#language en-US "My Dynamic Page"
#language fr-FR "My Dynamic Page Spanish"
#language en-US "My subtitle text"
#language fr-FR "Mi texto del subtítulo"
#string STR_SUBTITLE_TEXT
#language en-US " "
#language fr-FR " "
#string STR_SUBTITLE_TEXT2
#language en-US " "
#language fr-FR " "
#string STR_CPU_STRING
#language en-US "My CPU Speed is "
#language fr-FR "My CPU Speed is "
#string STR_CPU_STRING2
#language en-US " "
#language fr-FR " "

```

Example 137—Example of a Unicode string file

6. Determine if the driver must be localized: Does it need to support more than one language? If so, the strings must be translated. Determine if the languages can be displayed using the Latin-1 character set (European). If not, obtain fonts for the characters in the languages the driver supports.

At this point, a .uni file, a .vfr file, and a .h file have been produced. Typically, there is only one .uni file and one .vfr file per driver. More than one .uni file may be required if the driver presents multiple forms or menus. More than one .vfr file may be required in certain circumstances, for example, to simplify maintenance by holding an area of functionality in a separate .vfr file that changes often.

7. Implement HII Config Access Protocol to retrieve and save configuration information associated with the HII forms. The implementation of the HII Config Access Protocol is typically found in the file HiiConfigAccess.c. [Appendix A](#) contains a template for a HiiConfigAccess.c file for a UEFI Driver. The Config Access Protocol contains three services: ExtractConfig(), RouteConfig(), and DriverCallback(). The following example shows the definition of the HII Config Access Protocol for reference. When the HII setup browser is called, these functions are used to retrieve and store configuration setting as well as to retrieve default settings.

Note: *This is still the init section. The driver has not attached to those protocols yet.*

```

typedef struct _EFI_HII_CONFIG_ACCESS_PROTOCOL  EFI_HII_CONFIG_ACCESS_PROTOCOL;

///
/// This protocol provides a callable interface between the HII and
/// drivers. Only drivers which provide IFR data to HII are required
/// to publish this protocol.
///
struct _EFI_HII_CONFIG_ACCESS_PROTOCOL {
    EFI_HII_ACCESS_EXTRACT_CONFIG      ExtractConfig;
    EFI_HII_ACCESS_ROUTE_CONFIG        RouteConfig;
    EFI_HII_ACCESS_FORM_CALLBACK       Callback;
};

extern EFI_GUID gEfiHiiConfigAccessProtocolGuid;

```

Example 138—Example of a Unicode string file

8. Register all the packages from the driver entry point following the example in [Chapter 7](#) on adding HII packages.
9. If the UEFI Driver does not follow the UEFI Driver Model, install the HII Config Access Protocol from the driver entry point following the example in [Chapter 7](#). If the UEFI Driver does follow the UEFI Driver Model, the HII Config Access Protocol is installed in the Driver Binding Protocol Start() function on each handle the UEFI Driver manages and provides configuration.

At this point, the driver's init part is done. When the form is displayed to the user, the calls to the HII Config Access Protocol are made to retrieve and save configuration settings. It is up to the implementation of the HII Config Access Protocol to store configuration settings in NVRAM so they are available the next time the platform boots.

12.3 HII Protocols

Some protocols must be clearly understood in order to successfully implement a UEFI driver with HII functionality. The basic protocols consist of four consumable protocols and the HII Config Access Protocol produced by a UEFI Driver. They need not be used in any particular order. The `MdeModulePkg` provides the `UefiHiiServicesLib` that automatically looks up consumed HII protocols and makes them available to a UEFI Driver requiring the services they provide. A UEFI platform is not required to produce all of these protocols. The following is the list of protocols and the global variable provided by the `UefiHiiServicesLib`. If a global variable is set to `NULL`, it means that the platform does not produce that specific protocol. UEFI Drivers must handle all platform configurations, so it is important for a UEFI Driver to continue to function both when an HII related protocol is present and when an HII related protocol is absent.

10.	<code>EFI_HII_DATABASE_PROTOCOL</code>	<code>gHiiDatabase</code>
11.	<code>EFI_HII_STRING_PROTOCOL</code>	<code>gHiiString</code>
12.	<code>EFI_HII_FONT_PROTOCOL</code>	<code>gHiiFont</code>
13.	<code>EFI_HII_IMAGE_PROTOCOL</code>	<code>gHiiImage</code>

12.3.1 HII Database Protocol and HII String Protocol

Use the database protocol to submit the package of strings, fonts, forms, and so on, to the HII database. It is the most important of the HII protocols. Because the package is created at build time, and most of the package does not change, the driver does not have to call much later. *This can significantly speed up boot time.*

The strings protocol allows the general purpose forms to adapt to the configuration of a specific platform. This includes configuration information typed in by the user. The forms themselves are created by the VFR during build.

Basically, the string and database protocols facilitate database and string management. The browser simply gets things out of the database after the driver uses the set functions to put data into the database. The browser doesn't need to know how to parse the database or even know how strings are stored; it needs to know only how to parse the forms.

The EDK II provides a library, in the `MdeModulePkg`, called `HiiLib` that provides helper functions to simplify the use of the HII Database and HII String protocols. It also provides services to dynamically generate forms.

12.3.1.1 HII Database Protocol

HII data is contained in HII packages. For example, A driver might have a string package, a form package, and a small font package. HII supports package lists as a way to combine HII packages to create a single data structure for all the user interface HII data necessary for the driver. Rather than requiring the driver to split the packs up to, for example, provide the string pack to the string protocol and the font pack to the font protocol, the HII Database Protocol consumes the entire package list and portions it out to the various parts of the HII database. The package list format is described in the Human Interface Infrastructure Overview chapter of the *UEFI Specification*.

When a package list is submitted to the database (via `NewPackageList`), an ID, known as an HII handle, is associated with the data. This handle is required to manipulate the pack list's data to ensure uniqueness. For example, if two drivers submit string packs to the database, each have a string with an ID of 1 but they are different. The handle indicates which string with an ID of 1 to access.

One parameter to `NewPackageList` deserves special attention: `DriverHandle`. The driver handle indicates the handle on which the driver has put an instance of the `CONFIGURATION_ACCESS_PROTOCOL`. This protocol is used to obtain ("extract") the current configuration of a driver and to provide new configurations to it.

`UpdatePackageList` allows a driver to associate more than one package list with the same handle. This may simplify complex configurations by splitting the package into a common piece and additional configurations depending upon the cards SKU.

The Database protocol also supports methods to extract pieces from the database up to and including the entire database as well as `ListPackageLists` and `ExportPackageLists`. These functions are rarely useful for a driver but are the mechanisms by which the system places the HII data into the system table and also how the Setup browser obtains the data used to present its screens. The database protocol also supports notification functions for consumers of database data so they can determine if new packages have been added or existing ones removed.

Questions commonly asked include: Why are there individual protocols for some package types? Why isn't there a single protocol? The main reason is that the number of functions required became unwieldy. A secondary reason is that, for some smaller implementations, subsets of HII could be implemented. In reality this has not occurred.

The keyboard packages were judged as being simple enough to leave in the database protocol.

Keyboards are abstracted using a data structure per key. Each data structure defines the key code to which the data structure refers, as well as the unmodified Unicode weight and the weights when modified with Shift, Alt, and Shift + Alt. Only the keys that vary from the standard US English layout need be specified. Certain keys, such as NumLock, may also be assigned special functions.

12.3.1.2 HII String Protocol

The String Protocol consumes string packs. It also allows manipulation of strings already in the database, even if they were submitted via the database protocol.

It is quite common for a driver to need to manipulate certain strings when its data is in the HII database. Consider the case of a media card with attached mass storage devices. When the driver for the media card is created, the identification data of the mass storage devices attached aren't known. That data is derived when the card's driver is invoked, generally at `Start()`.

If the driver is to provide the mass storage device types to the setup browser, it is common to allocate empty strings so the build allocates string IDs to the strings. The driver can then parse the string pack to modify strings updating them with the drive id data itself and then submit the string pack. This is complex and tedious because the string packs are stored to be space efficient, not to be easily accessible. The String protocol already knows how to parse the string pack, however, and does provide methods to modify strings by ID. This makes the job of updating strings for dynamically derived data an easy one. Simply submit the string packs to the database, then modify the few strings that change dynamically. Blank strings can be checked for in IFR so empty channels don't have to be displayed.

12.3.1.3 Adding data to the HII database at boot time

There is more than one way to add information to the database. A crude way of adding information to the HII database is by using individual protocols to specify the fonts, strings, and forms. A better way is to use the HII Database Protocol. This protocol provides services to register the strings pack, fonts pack, forms pack, and so on, all at once. Because most of the package is static data, the driver does not have to do much work later during boot.

Note: *If the VFR compiler is used as part of the build, the package created may be published with this protocol.*

Also, note that the database is not complete at build time. The driver cannot know all the data it needs about the end-user's specific system hardware or other devices connected to the hardware. For example, the driver can't know a specific platform's MAC address at build time, which specific mass storage devices are attached via SCSI, each mass storage device's version information, and so on. That type of information is acquired during setup. During setup or boot, the package for the HII database must be updated.

Although data may be modified before being submitted to the database, that process is both difficult and convoluted. Use the `SetString()` function in the HII String Protocol instead.

For configurable data, or for data not available at build time, use a question mark in the package for each of the blank fields. During boot, the driver requests that information. Use the set string functionality of the HII String Protocol to specify the ID of the new package list and update the database with the new string from the build file.

Note: *If driver A creates a package list for the database, and another driver B creates another package list for the database, driver A's string #12 is not the same as driver B's string #12.*

12.3.1.4 Update the database via the byte offset of a configurable field

To modify a form after build-time, include a comment line (a macro for the VFR compiler) in the form's source code. The comment line does not generate code in the form. It simply indicates the byte offset of the value which does change the platform-specific information. The driver does not need to know how to parse the whole form to find that value. Instead, a driver can use the offset to find out where to edit the form.

Take the example of a SCSI driver with 2 drives specified as the default. In this example, the end-user platform actually has 3 drives. The driver searches for the appropriate comment to find the offset and the compiler tells the driver that the description of the logical unit is at line 437. The driver goes to that location, adds new forms data for the third drive and "slides" the rest of the configuration forms down. Essentially, new data is inserted into a newly created hole. Because the Internal Forms Language (IFL) is decision-independent, there are no fixed addresses in the code so data may be moved from one location to another relatively easily. The IFL also uses names for references, not pointers. For example, if 20 bytes of data need to be added at location 437, the 20 bytes can be copied into the new form.

Note: *The driver can do a get operation on the whole form or on just the string. The driver can do a get operation on the string because it uses the existing infrastructure (the platform's browser and other tools), which already know how to parse the database to find the appropriate data.*

12.3.1.5 Using strings to create forms as-needed

Use strings to create forms as needed. For example, most of the time, a SCSI has only 2 drives, but could have up to 8. Instead of creating a static form with 8 fields, and only 2 filled at boot time, a form with the 2 required fields can be created dynamically. The other 6 unused fields would not be displayed until they are actually needed.

12.3.1.6 Using strings to modify forms

In general, about 80% of any given form is static and common across the system's hardware. The other 20% is specific to that platform.

When adding information after build, it is sometimes easier to simply update a form. Other times it's easier to create a new form and turn it in. In general, a new form should be created if 70-80% of the information is new or has changed.

The VFR programming language explains how to work with forms and includes tips and suggestions for modifying forms.

12.3.1.7 HII Database Protocol with Export Package List

The HII Database Protocol provides a service to export all registered packages into an Export Package List. This includes packages registered by all UEFI Drivers. The Export Package List is not typically used by UEFI Drivers themselves. Instead, its purpose is to provide a single interface for external entities to extract the data needed to configure the system remotely.

Note: Programs that perform remote configuration do not have access to callbacks so questions related to callbacks are not visible remotely. Requests to read and write configuration data are routed to HII Config Access Protocol instances.

12.3.2 HII Config Routing Protocol

The Configuration Routing Protocol is not used by UEFI Drivers. However, it is important to understand its role in the configuration process. This protocol is used by consumers of forms to determine the current configuration of the tags (questions) associated with the forms and to change the configuration of the corresponding data.

The data format for both output from the drivers and input into the drivers is Unicode strings of ampersand separated name=value pairs. Each string is associated with a particular form, and hence, a particular driver. Specific name=value pairs at the start of a string of data associate the data with a particular instance of a driver.

This format can seem a little cumbersome at times but does provide a common, well defined mechanism to present the data. It is useful particularly in cases where the configuration of the system is handled remotely. It is also useful in cases where the same configuration data is applied to multiple systems, such as when systems are initially received by an IT department.

The data provided by the driver includes the leading name=value pairs. The data provided by the configuration program consists of a single string that may be consumed by multiple drivers (hence the name multi-config string). The routing protocol uses the leading name=value pairs to break-up the multi-config string and to determine the correct consumer of each of the substrings. Each driver receives only its own configuration data via the HII Config Access Protocol described below.

The leading name=value pairs (all in upper case only) are:

- GUID – The GUID in the Setup Form associated with this data
- NAME – The name of the driver
- PATH – The binary device path to the driver's device

A UEFI Driver may describe not only the current configuration but also several alternate configurations. Each alternate configuration is described by an identifier and preceded by a name=value pair with the name ALTCFG and the value indicating the alternate configuration in the Form. These are typically default configurations.

A UEFI driver maps its configuration into an array that is also represented as a C data structure. In this case, each configurable item is represented by three consecutive name=value pairs:

- OFFSET—The byte offset into the structure of the item
- WIDTH—The number of bytes the item consumes
- VALUE—The current (or new) configuration of the item

Helper functions map the string into a memory array to be stored by the UEFI Driver.

A UEFI Driver may receive a request for only certain configuration values, in which case only the names (and not the = or value) are filled in. The driver must fill in the values for the requested names.

If a UEFI Driver receives a configuration string containing incorrect leading name=value pairs, unknown names or out of bound values, the driver must reject the configuration request. In other words, the driver always validates the input string.

There is no requirement to include all name=value pairs in a configuration change string. The configuration associated with all names not mentioned in the string should not change. The UEFI Driver must ensure that the results of the reconfiguration are valid.

A UEFI Driver must provide a name=value pair parser that is tolerant of different formats of numbers—0ab, ab, and AB are all the same number. Similarly, the parser must be tolerant of case changes in names—Fred=5, fred=5, and FRED=5 should all be tolerated.

A UEFI Driver implementation of the HII Config Access Protocol must pay close attention to the memory allocation and deallocation requirements of the HII Config Access Protocol. Sometimes, the caller allocates the memory, other times, the callee allocates the memory. See the EFI HII Configuration Access Protocol section of the *UEFI Specification* for more details.

12.3.2.1 Remote configuration

Previously, even when configuration is local, every PC BIOS legacy option ROM had to carry its own setup—this took up a lot of space. With HII, only the platform needs to carry the browser. The driver carries only the package—the fonts and strings that the browser doesn't know about. For drivers having a significant amount of configuration, using HII functionality can help reduce the driver's size by as much as 20% or 30%.

For example, a platform may require some configuration at runtime. Or a platform may require remote configuration by an Information Technology (IT) administrator at a remote server that allows them to configure some settings, and send those settings back. In order to do this, the management application typically wants the entire database of information. Such an application sends that database off to the remote system, which does the configuration via its own setup browser, then sends the data back. The management application provides configuration changes to the platform that are routed back to the UEFI Driver managing the device being configured. This means a driver can support remote configuration without having to implement all the functions that the browser and management application already provide.

Note: *Configuration data, whether configuration is remote or local, does not need to use callbacks. In fact, a remote browser ignores all the pieces of a form involving callbacks. Once the configuration is on the end-user platform, callbacks are functional again because they are on a local machine.*

12.3.3 HII Config Access Protocol

The HII Config Access Protocol is produced by the UEFI driver. It has three key functions that are published to the HII database. The first two functions are used by the Configuration Routing Protocol to extract data from drivers and to provide configuration back to the drivers. The format of the configuration is modeled after the

CGI: A Unicode string of ampersand separated name = value pairs ($x=1&y=2&z=3A$). The names and values are specified in the forms. Only names are provided for extract requests. The driver cannot assume that all names in a form are present in a request—the caller may limit the entries to only those it needs.

Callbacks are the method by which the browser and driver directly communicate with each other. The forms describe when to invoke callbacks and they provide some context for the callback.

Use callbacks to update dynamic data, such as ambient temperature, fan speed, etc. They should not be used to modify how items are displayed.

The following three key functions are published to the HII database:

- **ExtractConfig function:** This function is called by the HII engine at the beginning of a particular form. This function gives the driver a chance to perform tasks before the form is processed by the HII database engine. For example, the function could test the current NVRAM data structure to make sure it is not corrupt. This function also allows the browser to display current configuration information.

Note: *The ExtractConfig function eliminates the need to use the previous, tedious method of manually outputting to the console, reading strings back from the console, and manually interpreting those strings.*

- **RouteConfig function:** This function allows the browser to obtain and change configuration information upon the exit of the form. It performs the final store and routes the appropriate data out to whoever needs it. For example, this function copies the current data back to the data structure in NVRAM. This function processes any changes that the user enters.
- **Callback function:** This function is called when a user makes changes. After the changes are saved, the original data structure is updated with the new settings.

Note: *This is not a callback in the traditional sense. This function is used by the browser to route data back to the appropriate driver so each driver can process its own configuration.*

12.3.3.1 Sample code for routing protocols

The following three examples show how the ExtractConfig, RouteConfig, and Callback functions of the Config Access Protocol may be used.

```

EFI_STATUS
EFI API
ExtractConfig (
    IN CONST EFI_HII_CONFIG_ACCESS_PROTOCOL *This,
    IN CONST EFI_STRING Request,
    OUT EFI_STRING *Progress,
    OUT EFI_STRING *Results
)
{
    EFI_STATUS Status;
    UINTN BufferSize;
    DRIVER_SAMPLE_PRIVATE_DATA *PrivateData;
    EFI_HII_CONFIG_ROUTING_PROTOCOL *HiiConfigRouting;
    EFI_STRING ConfigRequest;
    EFI_STRING ConfigRequestHdr;
}

```

```

UINTN                                     Size;
BOOLEAN                                    AllocatedRequest;

if (Progress == NULL || Results == NULL) {
    return EFI_INVALID_PARAMETER;
}
// Initialize the local variables.
ConfigRequestHdr = NULL;
ConfigRequest    = NULL;
Size             = 0;
*Progress        = Request;
AllocatedRequest = FALSE;

PrivateData = DRIVER_SAMPLE_PRIVATE_FROM_THIS (This);
HiiConfigRouting = PrivateData->HiiConfigRouting;

// Get Buffer Storage data from EFI variable.
// Try to get the current setting from variable.
BufferSize = sizeof (DRIVER_SAMPLE_CONFIGURATION);
Status = gRT->GetVariable (
    VariableName,
    &mFormSetGuid,
    NULL,
    &BufferSize,
    &PrivateData->Configuration
);
if (EFI_ERROR (Status)) {
    return EFI_NOT_FOUND;
}

if (Request == NULL) {
    // Request is set to NULL, construct full request string.
    //
    // Allocate and fill a buffer large enough to hold the <ConfigHdr> template
    // followed by "&OFFSET=0&WIDTH=%016LX" followed by a Null-terminator
    ConfigRequestHdr = HiiConstructConfigHdr (
        &mFormSetGuid,
        VariableName,
        PrivateData->DriverHandle[0]
    );
    Size = (StrLen (ConfigRequestHdr) + 32 + 1) * sizeof (CHAR16);
    ConfigRequest = AllocateZeroPool (Size);
    ASSERT (ConfigRequest != NULL);

    AllocatedRequest = TRUE;
    UnicodeSPrint (
        ConfigRequest,
        Size,
        L"%s&OFFSET=0&WIDTH=%016LX",
        ConfigRequestHdr,
        (UINT64)BufferSize
    );
    FreePool (ConfigRequestHdr);
} else {
    // Check routing data in <ConfigHdr>.
    // Note: if only one Storage is used, then this checking could be skipped.
    if (!HiiIsConfigHdrMatch (Request, &mFormSetGuid, NULL)) {
        return EFI_NOT_FOUND;
    }
    // Set Request to the unified request string.
    ConfigRequest = Request;
    // Convert buffer data to <ConfigResp> by helper function BlockToConfig()
    Status = HiiConfigRouting->BlockToConfig (
        HiiConfigRouting,
        ConfigRequest,
        (UINT8 *) &PrivateData->Configuration,
        BufferSize,

```

```

        Results,
        Progress
    );
}

// Free the allocated config request string.
if (AllocatedRequest) {
    FreePool (ConfigRequest);

    // Set Progress string to the original request string.
    if (Request == NULL) {
        *Progress = NULL;
    } else if (StrStr (Request, L"OFFSET") == NULL) {
        *Progress = Request + StrLen (Request);
    }
    return Status;
}

. .

return EFI_SUCESS
}

```

Example 139—ExtractConfig() Function

```

EFI_STATUS
EFIAPI
RouteConfig (
    IN CONST EFI_HII_CONFIG_ACCESS_PROTOCOL  *This,
    IN CONST EFI_STRING                      Configuration,
    OUT EFI_STRING                           *Progress
)
{
    EFI_STATUS          Status;
    UINTN              BufferSize;
    DRIVER_SAMPLE_PRIVATE_DATA   *PrivateData;
    EFI_HII_CONFIG_ROUTING_PROTOCOL *HiiConfigRouting;

    . .

    if (Configuration == NULL || Progress == NULL) {
        return EFI_INVALID_PARAMETER;
    }
    PrivateData = DRIVER_SAMPLE_PRIVATE_FROM_THIS (This);
    HiiConfigRouting = PrivateData->HiiConfigRouting;
    *Progress = Configuration;

    // Check routing data in <ConfigHdr>.
    // Note: if only one Storage is used, then this checking could be
    // skipped.
    if (!HiiIsConfigHdrMatch (Configuration, &mFormSetGuid, NULL)) {
        return EFI_NOT_FOUND;
    }

    //
    // Get Buffer Storage data from EFI variable
    //
    BufferSize = sizeof (DRIVER_SAMPLE_CONFIGURATION);
    Status = gRT->GetVariable (
        VariableName,
        &mFormSetGuid,
        NULL,
        &BufferSize,
        &PrivateData->Configuration
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }
}

```

```

    }
    // Convert <ConfigResp> to buffer data by helper function ConfigToBlock()
    BufferSize = sizeof (DRIVER_SAMPLE_CONFIGURATION);

    Status = HiiConfigRouting->ConfigToBlock (
        HiiConfigRouting,
        Configuration,
        (UINT8 *) &PrivateData->Configuration,
        &BufferSize,
        Progress
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }
    // Store Buffer Storage back to EFI variable
    Status = gRT->SetVariable(
        VariableName,
        &mFormSetGuid,
        EFI_VARIABLE_NON_VOLATILE |
        EFI_VARIABLE_BOOTSERVICE_ACCESS,
        sizeof (DRIVER_SAMPLE_CONFIGURATION),
        &PrivateData->Configuration
    );
    return Status;
}

```

Example 140—RouteConfig() Function

```

EFI_STATUS
EFIAPI
DriverCallback (
    IN CONST EFI_HII_CONFIG_ACCESS_PROTOCOL *This,
    IN EFI_BROWSER_ACTION Action,
    IN EFI_QUESTION_ID QuestionId,
    IN UINT8 Type,
    IN EFI_IFR_TYPE_VALUE *Value,
    OUT EFI_BROWSER_ACTION_REQUEST *ActionRequest
)
{
    DRIVER_SAMPLE_PRIVATE_DATA *PrivateData;
    EFI_STATUS Status;

    if ((Value == NULL) || (ActionRequest == NULL)) {
        return EFI_INVALID_PARAMETER;
    }
    Status = EFI_SUCCESS;
    PrivateData = DRIVER_SAMPLE_PRIVATE_FROM_THIS (This);
    switch (QuestionId) {
    case 0x1234:
        // do some code
        break;
    default:
        break;
    }
    return Status;
}

```

Example 141—Callback function

12.3.4 Rarely used HII protocols

There are two rarely used HII protocols: HII Font Protocol, and the HII Image Protocol. Though rarely used, understanding them is important.

12.3.4.1 HII Font Protocol

The HII Font Protocol provides functionality equivalent to the String Protocol but manages fonts instead. Fonts consist of glyphs, bit-mapped representations of characters. The characters are referred to by their Unicode *weight*, which is to say their corresponding binary value. For example, weight 0x0030 is a "0" (zero). A font is a series of glyphs bound together by name, size and similar visual characteristics.

The default font is the system font, which is 8x16 and 16x16 (for wide characters). Latin-1 characters in this standard font are provided by the system firmware. If a driver uses other characters, including e.g. Chinese, Korean, Hindi, Arabic, Hebrew, etc. A driver must provide all of the characters it uses. The build tools determine the actual characters used. Other fonts are identified by GUID.

TIP: It is strongly recommended that the system font be used for reasons of size and consistency.

Unlike strings, fonts are not separated by handle. When a driver provides fonts to the database, the new glyphs are merged with existing glyphs, provided that they are the same font. This means the display of a driver's data may use a different driver's font characters.

12.3.4.2 HII Image Protocol

HII provides simple support for images like graphical pictures and simplistic animation. There is no requirement for browsers to support graphics. The browser in EDK II does not support graphics and most setup browsers do not support graphics simply because of size requirements. The exception is for splash screens (banners).

12.4 HII functionality

HII functionality offers several benefits. One of the biggest is that HII functionality takes advantage of the platform's existing browser to standardize forms and change the way data is presented to the user. UEFI Drivers no longer need to include a browser and this simplifies drivers, helps reduce driver size, and helps standardize the interface for users. Also, because the forms support language localization, the driver no longer needs to manually manage language strings. Instead, the HII interface displays forms as appropriate for the languages specified by the driver writer.

12.4.1 Branding, and displaying a banner

HII makes it easier for vendors to brand their drivers. This includes displaying a unique splash screen or banner. This is done through HII forms. However, the forms themselves are defined in the VFR (visual forms representation) programming language. (See the VFR Programming Language Specification).

12.4.2 Specifying supported languages

The HII String Protocol allows strings and tokens to be used to specify the supported languages for a driver. The strings themselves are defined in a separate string file. That file is then published to the HII Database.

The string file must have at least one language definition and at least one string. If there is only one language specified, that language is the default. If more than one language is specified, then the first language listed is always the default language.

Note: *It is possible that no languages supported by the system are supported by the driver. In this case the browser selects the default language and proceeds. It is important to use the secondary language feature in HII to describe alternate languages to provide maximum flexibility for a set of strings.*

The following snippet from a Unicode string file shows American English (en-US) as the default language because it is first in the list. The string file includes support for two additional languages, French-Canadian (fr-CA), and British English (en-UK).

```
#langdef en-US "English"
#langdef fr-FR "Francais"
#langdef en-UK "British"

$string STR_INV_FORM_SET_TITLE #language en-US "ABC Information Sample"
                                #language fr-FR "Mi motor Espade arreglo"
                                #language en-UK "ABC Information Sample"
```

Example 142—Unicode string file with support for multiple languages

Note: *It costs the driver almost no processing time to support multiple languages because language selection is determined at the system level. However, adding support for multiple languages with additional strings and tokens can increase the size of the driver slightly. Adding support for many languages (for example, 100 or more) could increase the size of the driver more significantly.*

12.4.3 Specifying configuration information

HII functionality makes it easier to publish configuration information to a database. With HII functionality, the driver writer specifies the form layout for configuration information. The form layout points to static strings, as well as to data that is configurable by the user. The driver writer also defines the data structure of configurable data stored in NVRAM.

The strings are defined in a Unicode file (files with a .uni extension). During the driver's init section, the driver publishes the list of strings (such as language strings) and forms to the HII database with the HII handler. The driver also publishes its configuration routing protocols. The actual data structure of strings and forms is created as part of the build process.

The build tools take the .Uni file and the .Vfr file and produce a data structure. That data structure is stored in the HII database. Configurable data is stored in NVRAM.

When the HII engine is invoked, it runs the forms, pulls the strings it needs from the string database, and pulls the configurable settings it needs from NVRAM.

12.4.3.1 Using forms

Prior to HII, there was no standardized way to create forms. Instead, forms were created manually, and were manually output to the console. HII provides a standard way to create forms, making it easier to display information. Because HII functionality is standardized via forms, the driver no longer needs to manage the way users enter data, or worry about parsing the data. The HII engine parses the data to make sure it is appropriate for the defined field. See the discussion earlier in this section entitled "General Steps for Implementing HII Functionality."

To create forms, a UEFI Driver with HII functionality should use the VFR programming language and IFR defined in the Human Interface Infrastructure Overview chapter of the *UEFI Specification*. Refer to the *VFR Programming Language* for information about creating forms. The `MdeModulePkg` also contains a sample driver in the paths `MdeModulePkg/Universal/DriverSampleDxe` and `MdeModulePkg/Universal/HiiResourcesSampleDxe` that show example usages of VFR constructs.

12.4.3.2 Storing configuration information in nonvolatile storage

A UEFI Driver should store its configurable information in nonvolatile storage (NVRAM). This configuration information should be stored with the device so 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. Typically, drivers use the services of a bus I/O protocol to access the resources of 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 `GetVariable()` and `SetVariable()` to store configuration information.

12.4.4 Making configuration data available to other drivers

Configuration data is stored in NVRAM. The data structures that contain the static and configurable data for the driver are typically part of the package published to the HII database. In order to make configuration data available to other drivers, make sure to do the following:

1. Extract the forms from the form database.
2. Parse the forms for the names of the configurable options.
3. Use the HII Config Access Protocol to extract the data from all drivers.

12.4.4.1 Check validity of configuration options for a specific device

The *UEFI Specification* defines a `callBack()` service in the HII Config Access Protocol. This protocol interfaces with the VFR language. The callback protocol includes an action, `QuestionId`, type, value, and action request. When the user changes a configuration setting, this causes a call back to the driver. The driver then needs to check to see if the value entered is valid.

The data structure for configuration options is initialized via the driver's init entry point. The init reads the configuration data out of NVRAM and makes sure the data is valid. If any particular variable is invalid, the value for that variable is reset to its default.

The platform vendor can validate the configuration of all devices in the system before booting. In addition, the devices can be reset to their default configurations. If the firmware detects a corrupt configuration then a default configuration may be selected automatically. The platform vendor may choose to allow the user to select a menu item to force defaults on a specific device or all devices at once.

12.4.5 Check to see if configuration parameters are valid

To check configuration values and make sure they are valid, use the `ExtractConfig()` service of the HII Config Access Protocol. The HII setup browser uses this service to check for valid configuration values when the setup browser displays a form that was previously registered by the UEFI Driver. If the configuration values are not valid, then the setup browser may provide an option to reset the device to its default configuration settings. The default configuration settings may be retrieved using the `ExtractConfig()` service of the HII Config Access Protocol. This means the UEFI Driver that produces the HII Config Access Protocol must support requests for the current configuration settings as well as the default configuration settings.

12.5 Forms and VFR files

Here is a sample, simplified VFR file. It declares a form set with one form and uses a single variable store to retrieve and save configuration settings. The form contains a title and 4 questions.

1. Allows a user to select one of two pre-defined values.
2. Allows the user to type in a string value.
3. Allows the user to type in a numeric value.
4. Allows the user to select a button to reset settings to defaults.

```
/** @file
//
// Sample Setup formset.
//
// Copyright (c) 2004 - 2010, Intel Corporation. All rights reserved.<BR>
// 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.
//
//**/

#include "NVDataStruc.h"

formset
{
    guid    = FORMSET_GUID,
    title   = STRING_TOKEN(STR_FORM_SET_TITLE),
    help    = STRING_TOKEN(STR_FORM_SET_TITLE_HELP),
    classguid = EFI_HII_PLATFORM_SETUP_FORMSET_GUID,

    //
    // Define a Buffer Storage (EFI_IFR_VARSTORE)
    //
    varstore DRIVER_SAMPLE_CONFIGURATION, // This is the data structure type
    {
        varid = CONFIGURATION_VARSTORE_ID, // Optional VarStore ID
        name = MyIfrNVData, // Define referenced name in vfr
        guid = FORMSET_GUID; // GUID of this buffer storage
    }

    defaultstore MyStandardDefault,
    {
        prompt = STRING_TOKEN(STR_STANDARD_DEFAULT_PROMPT),
        attribute = 0x0000; // Default ID: 0000 standard default
    }

    defaultstore MyManufactureDefault,
    {
        prompt = STRING_TOKEN(STR_MANUFACTURE_DEFAULT_PROMPT),
        attribute = 0x0001; // Default ID: 0001 manufacture default
    }

    //
    // Define a Form (EFI_IFR_FORM)
    //
    form formid = 1, // Form ID
    {
        title = STRING_TOKEN(STR_FORM1_TITLE); // Form title
        subtitle text = STRING_TOKEN(STR_SUBTITLE_TEXT);
        subtitle text = STRING_TOKEN(STR_SUBTITLE_TEXT2);

        //
        // Define oneof (EFI_IFR_ONE_OF)
        //
        oneof name = MyOneOf, // Define reference name for Question
        {
            varid = MyIfrNVData.MyBaseAddress, // Use "DataStructure.Member" to
            prompt = STRING_TOKEN(STR_ONE_OF_PROMPT),
            help = STRING_TOKEN(STR_ONE_OF_HELP),
        }

        //
        // Define an option (EFI_IFR_ONE_OF_OPTION)
        //
        option text = STRING_TOKEN(STR_ONE_OF_TEXT1), value = 0x0, flags = 0;
        option text = STRING_TOKEN(STR_ONE_OF_TEXT2), value = 0x1, flags = 0;
        //
        // DEFAULT indicate that this option is to be marked with
        // EFI_IFR_OPTION_DEFAULT
        //
        option text = STRING_TOKEN(STR_ONE_OF_TEXT3), value = 0x2, flags = DEFAULT;
    } endoneof;

    //
    // Define a string (EFI_IFR_STRING)
    //
    string varid = MyIfrNVData.MyStringData,
    {
        prompt = STRING_TOKEN(STR_MY_STRING_PROMPT),
        help = STRING_TOKEN(STR_MY_STRING_HELP),
        flags = INTERACTIVE,
        key = 0x1236,
        minsize = 6,
        maxsize = 40,
    } endstring;
}

```

```

numeric varid = MyIfrNVData.MyHexData,
questionid = 0x1111,
prompt = STRING_TOKEN(STR_DATA_HEX_PROMPT),
help = STRING_TOKEN(STR_NUMERIC_HELP),
flags = DISPLAY_UINT_HEX | INTERACTIVE, // Display in HEX format (if not
                                         // specified, default is in decimal
                                         // format)
minimum = 0,
maximum = 250,
default = 175,
endnumeric;

resetbutton
defaultstore = MyStandardDefault,
prompt = STRING_TOKEN(STR_STANDARD_DEFAULT_PROMPT),
help = STRING_TOKEN(STR_STANDARD_DEFAULT_HELP),
endresetbutton;
endform;
endformset;

```

Example 143—Sample VFR file, simplified

12.6 HII Implementation Recommendations

12.6.1 Minimize callbacks

There are circumstances in which a callback is required. For example, callbacks are necessary when real-time data such as a temperature or voltage is required, or when direct password input is required to unlock a security feature.

However, the callback is useful for an extremely limited number of circumstances and can be used inappropriately.

Caution: *It is very important with UEFI drivers that the use of callbacks is minimized. The use of callbacks can significantly slow down a browser. Callbacks tend to be hard to maintain and are also typically very buggy. They don't adapt well to various video forms, which becomes an issue for interoperability between different types of devices. Finally, they cannot be used remotely, which creates significant problems with remote management of drivers.*

There are a number of useful techniques to reduce the use of callbacks. For example, use the rich set of comparison and calculation operators in VFR to validate input rather than resorting to callbacks. Also, modify the IFR (the language into which VFR compiles) before handing the IFR to HII. This allows the IFR to be adapted to the state of the system as the driver finds it. For example, don't use callbacks to determine attacked devices. Instead, determine the devices when providing the HII and fill in the data into the VFR.

Note that the HII engine can also do some testing of values, such as for minimum and maximum limits—a callback is not required for these operations. Instead, these checks are incorporated into the VFR sources, and the HII engine checks perform the tests against the minimum and maximum values. String compares may also be performed without the use of a callback.

- TIP:** Use a callback only when absolutely required, and when no other methods are available to perform the task. Almost nothing should be a callback.
- TIP:** Use callbacks only for dynamically changing data. Do not use callbacks for static data.
- TIP:** Do not use callbacks to format tables or make the interface look nice.
- TIP:** Do not make assumptions about the way the data returned from the callback is displayed.

Basically, let the HII engine perform as much of the work as possible and rigorously minimize the use of callbacks.

12.6.1.1 Callbacks create issues with remote configuration

One of the biggest issues with remote configuration is the use of callbacks (see the previous discussion for more information). For example, if configuration changes must be made to thousands of systems at a remote site, callback functions cannot be used, because the remote systems may be powered down or otherwise unavailable.

- TIP:** Use a callback only when absolutely required.

12.6.1.2 Callbacks create issues with interoperability

Callbacks are also an issue with regards to interoperability of remote devices. For example, a server might have a 32x4 plasma display. A browser may be implemented for VFR to support a 32x4 display, but the callback functions typically do not function well between device types. If a UEFI Driver is intended to be used in remote configuration scenarios, then avoid the use of callbacks.

12.6.2 Don't reparse the package list

Space is very important in the firmware. Size can be reduced by reparsing the forms and package list. However, it is better to let the code that already does that kind of parsing perform this task. This code already exists in the platform, so there is no reason to add it to a driver. In fact, even the browser should call a `GetString()` function instead of parsing the string package itself.

- TIP:** Avoid writing code that parses the package list.
- TIP:** When in doubt, submit the package list, then the driver can call the get-string function and set-string function.

12.6.3 Concentrate on critical aspects of the driver

Often people focus on what they can easily see of a driver, which tends to be the browser, not the actual driver. However, with HII functionality, a driver no longer needs to include its own browser. Instead, the driver can take advantage of the platform's browser and other code already written and a part of the platform.

TIP: Concentrate on the important parts of the driver (what it does), not on the more visible, probably browser-related, aspects. A UEFI 2.x conformant driver uses the platform's existing browser anyway

12.6.4 Perform usability testing

Many developers do not perform usability testing on their forms. When implementing HII functionality, make sure to test for ease of use, readability of the fields and forms, and the logical flow of concepts from forms to sub-forms.

12.7 Porting to UEFI HII functionality

HII allows the platform's existing browser to be used to display and manage forms for user input. In doing so, HII functionality **replaces** or supplements older protocols:

- Driver Configuration Protocol and Driver Configuration 2 Protocol: If a UEFI Driver is required to only be compatible with the UEFI 2.1 Specification or higher, then **replace** the use of these protocols with HII functionality.
- Simple Text Input Protocol, Simple Text Output Protocol: UEFI Drivers, in general, are not allowed to use UEFI console protocols. The one exception is the Driver Configuration Protocol `SetOptions()` service. If a UEFI Driver is required to only be compatible with *UEFI 2.1 Specification* or higher, the Driver Configuration Protocols are not required and the Simple Text Input Protocol and Simple Text Output Protocol should not be used.
- Convert strings used by Driver Configuration Protocol `SetOptions()` to a .uni file.
- Convert questions and other user interactions in Driver Configuration Protocol `SetOptions()` to a .vfr file. Only use HII callbacks if absolutely required.
- Convert Driver Configuration Protocol `ForceDefaults()` functionality into .vfr sources.
- Convert Driver Configuration Protocol `OptionsValid()` functionality into .vfr sources.

UEFI Driver Diagnostics

The Driver Diagnostics Protocols are optional features that allow UEFI Drivers following the UEFI Driver Model to provide diagnostics for the devices under UEFI Driver management. Use of these protocols depends on the UEFI Driver Model concepts so Service Drivers, Root Bridge Drivers, and Initializing Drivers never produce the Driver Diagnostics Protocols.

The Driver Binding Protocol `start()` function may perform some quick checks of a device's status, but checks taking extended time to execute should be provided in a Driver Diagnostic Protocol implementation. Doing so improves the overall platform boot performance by deferring extensive diagnostics to a separate protocol not required to execute on every boot.

The Driver Diagnostics Protocol and the Driver Diagnostics 2 Protocol are very similar. The only difference lies in the type of language code used to specify the language for diagnostic result messages. The Driver Diagnostic Protocol uses ISO 639-2 language codes (i.e. `eng`, `fra`). The Driver Diagnostics 2 Protocol uses RFC 4646 language codes (i.e. `en`, `en-US`, `fr`). For diagnostics provided to platforms conforming to the *EFI 1.10 Specification*, use the Driver Diagnostics Protocol. For diagnostics provided to platforms conforming to the *UEFI 2.0 Specification* or above, use the Driver Diagnostics 2 Protocol. Since the only difference is the language code for the diagnostic message results, UEFI Drivers required to provide diagnostics typically produce both protocols so the two implementations can share the same diagnostic algorithms and diagnostic result messages.

The Driver Diagnostics Protocols are installed onto handles in the driver entry point of UEFI Drivers. [Chapter 7](#) provides details on the EDK II `UefiLib` library that provides helper functions to initialize UEFI Drivers following the UEFI Driver Model, including the installation of the Driver Diagnostics Protocols.

The Driver Diagnostic Protocols may be invoked from a UEFI Boot Manager if a platform provides those options to a user. A platform vendor can take advantage of Driver Diagnostic Protocol implementations for devices to improve overall system diagnostics for the user. These protocols may also be invoked through a UEFI Application that performs diagnostics.

Use the `drvdiag` command to test the functionality of Driver Diagnostic Protocol implementation and to diagnose issues on platforms that either build the UEFI Shell in or provide the ability to boot the UEFI Shell from a boot device. The `drvdiag` command provides the list of devices that support diagnostic operations and the ability to run diagnostics on a specific device and report the results.

If a controller is managed by more than one UEFI Driver, there may be multiple instances of the Driver Diagnostics Protocols that apply to a single controller. The consumers of the Driver Diagnostics Protocols have to decide how the multiple drivers supporting diagnostics are presented to users so they can select the desired diagnostic. For example, a PCI bus driver may produce the Driver Diagnostics Protocol to verify the

functionality of a specific PCI slot. The UEFI Driver for a SCSI adapter inserted into that same PCI slot may produce diagnostics for the SCSI host controller. Both sets of diagnostics may be useful to a user when testing the platform. The UEFI Shell `drvdiag` command does support this use case.

[Appendix B](#) contains a table of example drivers from the EDK II along with the features each implement. The EDK II provides example drivers with full implementations of the Driver Diagnostics Protocols.

Note: *The Driver Diagnostics Protocols are used rarely, and platform vendors may or may not invoke the Driver Diagnostics Protocols.*

13.1 Driver Diagnostics Protocol Implementations

The implementation of the Driver Diagnostics Protocols for a specific driver is typically found in the file `DriverDiagnostics.c`. [Appendix A](#) contains a template for `DriverDiagnostics.c`, a file for a UEFI Driver. This file typically contains the following:

- Add global variable for the `EFI_DRIVER_DIAGNOSTICS_PROTOCOL` instance to `DriverDiagnostics.c`.
- Add global variable for the `EFI_DRIVER_DIAGNOSTICS2_PROTOCOL` instance to `DriverDiagnostics.c`.
- Add Static table of diagnostics result messages as Unicode strings to `DriverDiagnostics.c`.
- Implementation of the `RunDiagnostics()` service
- Install all the Driver Diagnostics Protocols in the driver entry point.
- If the UEFI Driver supports the unload feature, uninstall all the Driver Diagnostics Protocols in the `Unload()` function.

The Driver Diagnostics Protocols provide diagnostics result messages in one or more languages. At a minimum, the protocols should support the English language. The Driver Diagnostic Protocol advertises the languages it supports in a data field called `SupportedLanguages`. This data field is a null-terminated ASCII string that contains one or more 3 character ISO 639-2 language codes with no separator character. The Driver Diagnostic 2 Protocol also advertises the languages it supports in a data field called `SupportedLanguages`. This data field is a null-terminated ASCII string that contains one or more RFC 4646 language codes separated by semicolons (';').

A consumer of the Driver Diagnostics Protocols may parse the `SupportedLanguages` data field to determine 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 Protocols to see if diagnostics result messages are available in the requested language.

[Example 144](#), below, shows the protocol interface structure for the Driver Diagnostic Protocol and the following [Example 145](#) shows the protocol interface structure for the Driver Diagnostics 2 Protocol for reference. Both are composed of one service called `RunDiagnostics()` and a data field called `SupportedLanguages`.

```

typedef struct _EFI_DRIVER_DIAGNOSTICS_PROTOCOL  EFI_DRIVER_DIAGNOSTICS_PROTOCOL;

///
/// Used to perform diagnostics on a controller that an EFI Driver is managing.
///
struct _EFI_DRIVER_DIAGNOSTICS_PROTOCOL {
    EFI_DRIVER_DIAGNOSTICS_RUN_DIAGNOSTICS  RunDiagnostics;
    ///
    /// A Null-terminated ASCII string that contains one or more ISO 639-2
    /// language codes. This is the list of language codes that this protocol
    /// supports.
    ///
    CHAR8                                *SupportedLanguages;
};

```

Example 144—Driver Diagnostics Protocol

```

typedef struct _EFI_DRIVER_DIAGNOSTICS2_PROTOCOL  EFI_DRIVER_DIAGNOSTICS2_PROTOCOL;

///
/// Used to perform diagnostics on a controller that an EFI Driver is managing.
///
struct _EFI_DRIVER_DIAGNOSTICS2_PROTOCOL {
    EFI_DRIVER_DIAGNOSTICS2_RUN_DIAGNOSTICS RunDiagnostics;
    ///
    /// A Null-terminated ASCII string that contains one or more RFC 4646
    /// language codes. This is the list of language codes that this protocol
    /// supports.
    ///
    CHAR8                                *SupportedLanguages;
};

```

Example 145—Driver Diagnostics 2 Protocol

UEFI Drivers declare global variables for the Driver Diagnostics Protocol and Driver Diagnostics 2 Protocol instances produced. The *SupportedLanguages* fields are typically initialized by the UEFI Driver in the declaration for the specific set of language the UEFI Driver supports. The example below shows how the Driver Diagnostics Protocols are typically declared in a driver, and in this case declared to support both English and French.

```

#include <Uefi.h>
#include <Protocol/DriverDiagnostics.h>
#include <Protocol/DriverDiagnostics2.h>

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_DIAGNOSTICS_PROTOCOL gAbcDriverDiagnostics = {
    (EFI_DRIVER_DIAGNOSTICS_RUN_DIAGNOSTICS) AbcRunDiagnostics,
    "engfra"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_DIAGNOSTICS2_PROTOCOL gAbcDriverDiagnostics2 = {
    AbcRunDiagnostics,
    "en;fr"
};

```

Example 146—Driver Diagnostics Protocol declaration

The implementations of the Driver Diagnostics Protocol change in complexity depending on the type of UEFI Driver Model driver. A device driver is the simplest 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 are discussed later in this chapter.

The `EFI_DRIVER_DIAGNOSTICS_PROTOCOL` and `EFI_DRIVER_DIAGNOSTICS2_PROTOCOL` are installed onto the driver's image handle. It is possible for a driver to produce more than one instance of the Driver Diagnostics Protocols. All additional instances of the Driver Diagnostics Protocols must be installed onto new handles.

The Driver Diagnostics Protocols can either be installed directly using the UEFI Boot Service `InstallMultipleProtocolInterfaces()`. However, the EDK II library `UefiLib` provides a number of helper functions to install the Driver Diagnostics Protocols. The helper functions that are covered in more detail in [Chapter 7](#) are:

- `EfiLibInstallAllDriverProtocols()`
- `EfiLibInstallAllDriverProtocols2()`

If an error is generated installing any of the Driver Diagnostics Protocol instances, then the entire driver should fail and return a error status such as `EFI_ABORTED`. If a UEFI Driver implements the `Unload()` feature, any Driver Diagnostics Protocol instances installed in the driver entry point must be uninstalled in the `Unload()` function.

The implementation of the Driver Diagnostics Protocols for a specific driver is typically found in the file `DriverDiagnostics.c`. This file contains the instances of the `EFI_DRIVER_DIAGNOSTICS_PROTOCOL` and `EFI_DRIVER_DIAGNOSTICS2_PROTOCOL` along with the implementation of `RunDiagnostics()`. [Appendix A](#) contains a template for a `DriverDiagnostics.c` file for a UEFI Driver.

13.2 RunDiagnostics() Implementations

The `RunDiagnostics()` service runs diagnostics on the controller a driver is managing or a child 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. The caller may choose to log the results or display them. The example below shows an empty implementation of the `RunDiagnostics()` service for the Driver Diagnostics 2 Protocol. The recommended implementation style shown allows the same `RunDiagnostics()` service implementation to be shared between the Driver Diagnostics Protocol and the Driver Diagnostics 2 Protocol.

```
#include <Uefi.h>
#include <Protocol/DriverDiagnostics2.h>

EFI_STATUS
EFIAPI
AbcRunDiagnostics (
    IN EFI_DRIVER_DIAGNOSTICS2_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 147—RunDiagnostics() Service

The *DiagnosticType* parameter tells the driver the type of diagnostics to perform. Standard diagnostics must be implemented and test basic functionality. They 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 manufacturing and test environments.

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 following 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 *AllocatePool()*, and it is the caller's responsibility to free this buffer with *FreePool()*.

The Driver Diagnostics Protocols are available only for devices a driver is currently managing. Because UEFI 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 must connect all drivers to all devices, so that the user can be shown all devices supporting diagnostics.

13.2.1 Device Drivers

Device drivers that implement *RunDiagnostics()* must verify that *ChildHandle* is **NULL** and that *ControllerHandle* represents a device that the device driver is currently managing. In addition, *RunDiagnostics()* must verify that the requested Language is in the set of languages that the UEFI Driver supports. The following example shows the steps required to check these parameters. If these checks pass, the diagnostic are executed and results are returned. In this specific example, the driver opens the PCI I/O Protocol in its Driver Binding *start()* function. This is why **gEfiPciIoProtocolGuid** is used in the call to the EDK II Library *UefiLib* function *EfiTestManagedDevice()* that checks to see if the UEFI Drivers that is providing this *RunDiagnostics()* service is currently managing *ControllerHandle*. If the private context structure is required, typically the UEFI Boot Service *OpenProtocol()* is used to open one of the UEFI Driver produced protocols on *ControllerHandle* and then use a *CR()* based macro to retrieve a pointer to the private context structure.

```

#include <Uefi.h>
#include <Protocol/DriverDiagnostics2.h>
#include <Protocol/PciIo.h>
#include <Library/BaseMemoryLib.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI

```

```

AbcRunDiagnostics (
    IN EFI_DRIVER_DIAGNOSTICS2_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
)
{
    EFI_STATUS Status;
    CHAR8 *SupportedLanguages;
    BOOLEAN Rfc4646Language;
    BOOLEAN Found;
    UINTN Index;

    //
    // ChildHandle must be NULL for a Device Driver
    //
    if (ChildHandle != NULL) {
        return EFI_UNSUPPORTED;
    }

    //
    // Make sure this driver is currently managing ControllerHandle
    //
    Status = EfiTestManagedDevice (
        ControllerHandle,
        gAbcDriverBinding.DriverBindingHandle,
        &gEfiPciIoProtocolGuid
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    //
    // Check input parameters
    //
    if (Language == NULL ||
        ErrorType == NULL ||
        BufferSize == NULL ||
        Buffer == NULL) {
        return EFI_INVALID_PARAMETER;
    }

    //
    // Make sure Language is in the set of Supported Languages
    //
    SupportedLanguages = This->SupportedLanguages;
    Rfc4646Language = (BOOLEAN)(This == &gAbcDriverDiagnostics2);
    Found = FALSE;
    while (*SupportedLanguages != 0) {
        if (Rfc4646Language) {
            for (Index = 0;
                SupportedLanguages[Index] != 0 && SupportedLanguages[Index] != ';';
                Index++);
            if ((AsciiStrnCmp(SupportedLanguages, Language, Index) == 0) &&
                (Language[Index] == 0)) {
                Found = TRUE;
                break;
            }
            SupportedLanguages += Index;
            for (; *SupportedLanguages != 0 && *SupportedLanguages == ';';
                SupportedLanguages++);
        } else {
            if (CompareMem (Language, SupportedLanguages, 3) == 0) {

```

```

        Found = TRUE;
        break;
    }
    SupportedLanguages += 3;
}
//
// If Language is not a member of SupportedLanguages, then return EFI_UNSUPPORTED
//
if (!Found) {
    return EFI_UNSUPPORTED;
}

//
// Perform Diagnostics Algorithm on ControllerHandle for the
// type of diagnostics requested in DiagnosticsType
//
// Return results in ErrorType, Buffer, and BufferSize
//
// If Diagnostics Algorithm fails, then return EFI_DEVICE_ERROR
//

return EFI_SUCCESS;
}

```

Example 148—RunDiagnostics() for a Device Driver

To verify the operation of the controller, diagnostic algorithms typically use the services of the protocols the driver produces and the services of the protocols the driver consumes. 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. Use the Block I/O Services to verify that the entire driver is working as expected.

13.2.2 Bus Drivers and Hybrid Drivers

Bus drivers and hybrid drivers implementing the Driver Diagnostics Protocols must verify that `ControllerHandle` and `ChildHandle` represent a device currently managed by the driver. In addition, `RunDiagnostics()` must verify that the requested `Language` is in the set of languages supported by the UEFI Driver. The [following example](#) shows the steps required to check these parameters and also retrieve the private context data structure. If the checks pass, the diagnostics are executed and results returned.

In this specific example, the driver opens the PCI I/O Protocol in its Driver Binding Start() function. This is why `gEfiPciIoProtocolGuid` is used in the call to the EDK II Library `UefiLib` function `EfiTestManagedDevice()`. It checks to see if the UEFI Drivers providing the `RunDiagnostics()` service is currently managing `ControllerHandle`. If the private context structure is required, then, typically, the UEFI Boot Service `OpenProtocol()` is used to open one of the protocols on `ControllerHandle` that the UEFI Driver produced and then uses a `CR()` based macro to retrieve a pointer to the private context structure. If diagnostics are being run on `ChildHandle`, a produced protocol on `ChildHandle` can be opened.

Note: *If `ChildHandle` is `NULL`, a request is made to run diagnostics on the bus controller. If `ChildHandle` is not `NULL`, a request is made to run diagnostics on a child controller managed by the UEFI Driver.*

```

#include <Uefi.h>
#include <Protocol/DriverDiagnostics2.h>
#include <Protocol/PciIo.h>
#include <Library/BaseMemoryLib.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI
AbcRunDiagnostics (
    IN EFI_DRIVER_DIAGNOSTICS2_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
)
{
    EFI_STATUS Status;
    CHAR8 *SupportedLanguages;
    BOOLEAN Rfc4646Language;
    BOOLEAN Found;
    UINTN Index;

    //
    // Make sure this driver is currently managing ControllerHandle
    //
    Status = EfiTestManagedDevice (
        ControllerHandle,
        gAbcDriverBinding.DriverBindingHandle,
        &gEfiPciIoProtocolGuid
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    //
    // If ChildHandle is not NULL, then make sure this driver produced ChildHandle
    //
    if (ChildHandle != NULL) {
        Status = EfiTestChildHandle (
            ControllerHandle,
            ChildHandle,
            &gEfiPciIoProtocolGuid
        );
        if (EFI_ERROR (Status)) {
            return Status;
        }
    }

    //
    // Check input parameters
    //
    if (Language == NULL ||
        ErrorType == NULL ||
        BufferSize == NULL ||
        Buffer == NULL ) {
        return EFI_INVALID_PARAMETER;
    }

    //
    // Make sure Language is in the set of Supported Languages
    //
    SupportedLanguages = This->SupportedLanguages;
    Rfc4646Language = (BOOLEAN)(This == &gAbcDriverDiagnostics2);
    Found = FALSE;
}

```

```

while (*SupportedLanguages != 0) {
    if (Rfc4646Language) {
        for (Index = 0;
            SupportedLanguages[Index] != 0 && SupportedLanguages[Index] != ';';
            Index++);
        if ((AsciiStrnCmp(SupportedLanguages, Language, Index) == 0) &&
            (Language[Index] == 0)) {
            Found = TRUE;
            break;
        }
        SupportedLanguages += Index;
        for (; *SupportedLanguages != 0 && *SupportedLanguages == ';';
            SupportedLanguages++);
    } else {
        if (CompareMem (Language, SupportedLanguages, 3) == 0) {
            Found = TRUE;
            break;
        }
        SupportedLanguages += 3;
    }
}
// If Language is not a member of SupportedLanguages, then return EFI_UNSUPPORTED
//
if (!Found) {
    return EFI_UNSUPPORTED;
}

if (ChildHandle == NULL) {
    //
    // Perform Diagnostics Algorithm on the bus controller specified
    // by ControllerHandle for the type of diagnostics requested in
    // DiagnosticsType
    //
    // Return results in ErrorType, Buffer, and BufferSize
    //
    // If Diagnostics Algorithm fails, then return EFI_DEVICE_ERROR
    //
} else {
    //
    // Perform Diagnostics Algorithm on child controller specified
    // by ChildHandle for the type of diagnostics requested in
    // DiagnosticsType
    //
    // Return results in ErrorType, Buffer, and BufferSize
    //
    // If Diagnostics Algorithm fails, then return EFI_DEVICE_ERROR
    //
}

return EFI_SUCCESS;
}

```

Example 149—RunDiagnostics() for a Bus Driver or Hybrid Driver

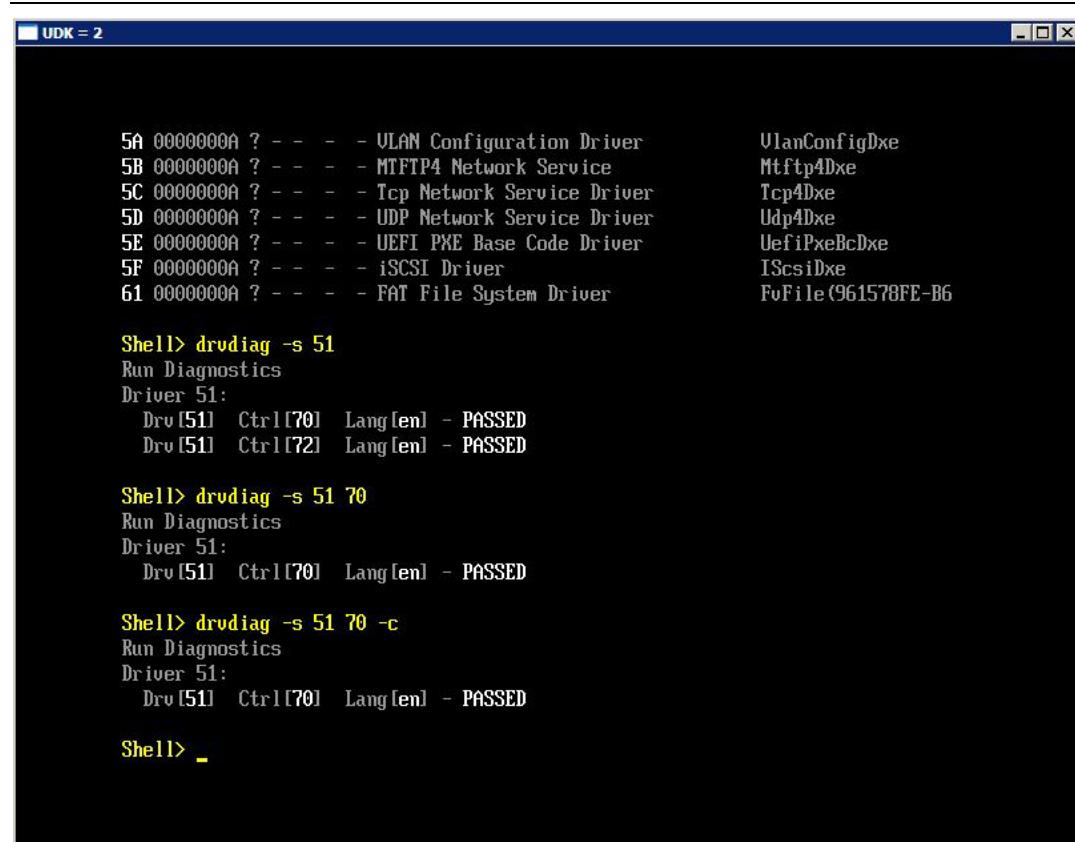
It is recommended that bus drivers and hybrid provide diagnostics for both the bus controller and the child controllers produced by these driver types. Implementing diagnostics for only the bus controller or only the child controllers is strongly discouraged.

13.2.3 RunDiagnostics() as a UEFI Application

One useful design aspect of the Driver Diagnostics Protocol is implementation of diagnostics as a UEFI application stored with a device (i.e. PCI Option ROM) or in an EFI System Partition. To do so, change the implementation of `RunDiagnostics()` so it does not directly execute the diagnostics, yet would perform the same parameter checks as before and still retrieve the private context data structure. Then, instead of executing diagnostic algorithms, use the UEFI Boot Service `LoadImage()` and the UEFI Boot Service `startImage()` to load and execute the UEFI application running the diagnostic algorithms. The application then returns the results of the diagnostics back to `RunDiagnostics()` and `RunDiagnostics()` returns the final results in the required format.

13.3 Testing Driver Diagnostics Protocols

Use the UEFI Shell command `drvdiag` to exercise the Driver Diagnostics Protocols. Run this command with no options to show the set of drivers the Driver Diagnostics Protocols support. The `drvdiag` command allows the different types of diagnostics tests to run on a controller, a specific child of a controller, or all the children of a controller. The [figure below](#) shows a few examples of using the UEFI Shell command `drvdiag` on the EDK II Nt32 platform to run diagnostics provided with the Block I/O driver for the Nt32 platform.



The screenshot shows a UDK Shell window titled "UDK = 2". The window displays a list of drivers and their corresponding UEFI GUIDs, followed by several "drvdiag" command executions.

GUID	Description	GUID
5A 0000000A ? - - -	ULAN Configuration Driver	VlanConfigDxe
5B 0000000A ? - - -	MTFTP4 Network Service	Mtftp4Dxe
5C 0000000A ? - - -	Tcp Network Service Driver	Tcp4Dxe
5D 0000000A ? - - -	UDP Network Service Driver	Udp4Dxe
5E 0000000A ? - - -	UEFI PXE Base Code Driver	UefiPxeBcDxe
5F 0000000A ? - - -	iSCSI Driver	IScsiDxe
61 0000000A ? - - -	FAT File System Driver	FvFile(961578FE-B6

Shell> **drvdiag -s 51**
Run Diagnostics
Driver 51:
 Drv [51] Ctrl [70] Lang [en] - PASSED
 Drv [51] Ctrl [72] Lang [en] - PASSED

Shell> **drvdiag -s 51 70**
Run Diagnostics
Driver 51:
 Drv [51] Ctrl [70] Lang [en] - PASSED

Shell> **drvdiag -s 51 70 -c**
Run Diagnostics
Driver 51:
 Drv [51] Ctrl [70] Lang [en] - PASSED

Shell> _

Figure 17—Testing Driver Diagnostics Protocols

Details on each UEFI Shell command used to test UEFI Drivers appear in [Chapter 31](#).

14

Driver Health Protocol

The Driver Health Protocol is a feature potentially required by UEFI Drivers following the UEFI Driver Model. If a UEFI Driver needs to report health status to the platform, provide warning or error messages to the user, perform length repair operations, or request that the user make hardware or software configuration changes, the Driver Health Protocol must be produced. This protocol is required only for devices potentially in a bad state and recoverable through either a repair operation or configuration change. The Driver Health Protocol should not be implemented if a device can never be in a bad state or a device can be in a bad state for which no remediation is possible.

The UEFI Boot Manager uses the services of the Driver Health Protocol, if present, to determine the health status of a device and display that status information on a UEFI console. The UEFI Boot Manager may also choose to perform actions to transition devices from a bad state to a usable state. See the EFI Driver Health Protocol section of the *UEFI Specification* for more details on how a UEFI Boot manager interacts with the Driver Health Protocol.

This chapter focuses on how to implement the Driver Health Protocol for a UEFI Driver managing a specific set of devices.

14.1 Driver Health Protocol Implementation

The implementation of the Driver Health Protocol is typically found in the file [DriverHealth.c](#). [Appendix A](#) contains a template for a `DriverHealth.c` file for a UEFI Driver. The list of tasks to implement the Driver Health Protocol feature follow:

- Add global variable for the `EFI_DRIVER_HEALTH_PROTOCOL` instance to `DriverHealth.c`.
- Add private fields, as required, to the design of the private context data structure that supports storing the current health status of a device and managing repair operations.
- Implement the `GetHealthStatus()` service of the Driver Health Protocol in `DriverHealth.c`.
- Implement the `Repair()` service of the Driver Health Protocol in `DriverHealth.c`.
- Install the Driver Health Protocol onto the same handle as that of the Driver Binding Protocol.
- If the UEFI Driver produces multiple Driver Binding Protocols, install the Driver Health Protocol on the same handles as those of the Driver Binding Protocol.
- If the UEFI Driver supports the unload feature, uninstall all the Driver Health Protocol instances in the `Unload()` function.

The example below shows the protocol interface structure for the Driver Health Protocol for reference and is composed of two services; `GetHealthStatus()` and `Repair()`.

```
typedef struct _EFI_DRIVER_HEALTH_PROTOCOL  EFI_DRIVER_HEALTH_PROTOCOL;  
  
///  
/// When installed, the Driver Health Protocol produces a collection of services  
/// that allow the health status for a controller to be retrieved. If a controller  
/// is not in a usable state, status messages may be reported to the user, repair  
/// operations can be invoked, and the user may be asked to make software and/or  
/// hardware configuration changes.  
///  
struct _EFI_DRIVER_HEALTH_PROTOCOL {  
    EFI_DRIVER_HEALTH_GET_HEALTH_STATUS  GetHealthStatus;  
    EFI_DRIVER_HEALTH_REPAIR             Repair;  
};
```

Example 150—Driver Health Protocol

This example declares a global variable called `gAbcDriverHealth` with the services `AbcGetHealthStatus()` and `AbcRepair()`. The UEFI Boot Service `InstallMultipleProtocolInterfaces()` is used to install the Driver Health Protocol instance `gAbcDriverHealth` onto the same `ImageHandle` as that of the Driver Binding Protocol instance `gAbcDriverBinding`.

```
#include <Uefi.h>  
#include <Protocol/DriverBinding.h>  
#include <Protocol/DriverHealth.h>  
#include <Library/UefiBootServicesTableLib.h>  
#include <Library/UefiLib.h>  
#include <Library/DebugLib.h>  
  
#define ABC_VERSION 0x10  
  
EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {  
    AbcSupported,  
    AbcStart,  
    AbcStop,  
    ABC_VERSION,  
    NULL,  
    NULL  
};  
  
GLOBAL_REMOVE_IF_UNREFERENCED  
EFI_DRIVER_HEALTH_PROTOCOL gAbcDriverHealth = {  
    AbcGetHealthStatus,  
    AbcRepair  
};  
  
EFI_STATUS  
EFIAPI  
AbcDriverEntryPoint (  
    IN EFI_HANDLE           ImageHandle,  
    IN EFI_SYSTEM_TABLE    *SystemTable  
)  
{  
    EFI_STATUS  Status;
```

```

//  

// Install driver model protocol(s) on ImageHandle  

//  

Status = EfiLibInstallDriverBinding (
    ImageHandle,           // ImageHandle
    SystemTable,          // SystemTable
    &gAbcDriverBinding,   // DriverBinding
    ImageHandle           // DriverBindingHandle
);
ASSERT_EFI_ERROR (Status);

//  

// Install Driver Family Override Protocol onto ImageHandle  

//  

Status = gBS->InstallMultipleProtocolInterfaces (
    &ImageHandle,
    &gEfiDriverHealthProtocolGuid,
    &gAbcDriverHealth,
    NULL
);
ASSERT_EFI_ERROR (Status);

return Status;
}

```

Example 151—Install Driver Health Protocol

14.2 GetHealthStatus() Implementations

The `GetHealthStatus()` service retrieves the health status for a controller a driver is managing or a child the driver has produced. This service is not allowed to use any of the console I/O related protocols. Instead, the health status information is returned to the caller. The caller may choose to log or display the health status information. The example below shows an empty implementation of the `GetHealthStatus()` service for the Driver Health Protocol.

```

#include <Uefi.h>
#include <Protocol/DriverHealth.h>

EFI_STATUS
EFIAPI
AbcGetHealthStatus (
    IN  EFI_DRIVER_HEALTH_PROTOCOL           *This,
    IN  EFI_HANDLE                           ControllerHandle OPTIONAL,
    IN  EFI_HANDLE                           ChildHandle   OPTIONAL,
    OUT EFI_DRIVER_HEALTH_STATUS            *HealthStatus,
    OUT EFI_DRIVER_HEALTH_HII_MESSAGE      **MessageList  OPTIONAL,
    OUT EFI_HII_HANDLE                      *FormHiiHandle OPTIONAL
)
{
}

```

Example 152—GetHealthStatus() Function of the Driver Health Protocol

HealthStatus is the return parameter reporting the status for the controller specified by *ControllerHandle* and *ChildHandle*. Descriptions of the various health status values returned in *HealthStatus* follow.

Table 23—Health Status Values

Health Status Name	Definition
EfiDriverHealthStatusHealthy	The controller is in a healthy state.
EfiDriverHealthStatusRepairRequired	The controller requires a repair operation taking an extended period of time to perform. The UEFI Boot Manager is required to call the Repair() function when this state is detected.
EfiDriverHealthStatusConfigurationRequired	The controller requires the user to make software or hardware configuration changes in order to put the controller into a healthy state. The set of software configuration changes are specified by the FormHiiHandle parameter. The EFI Boot Manager may call the EFI_FORM_BROWSER2_PROTOCOL.SendForm() function to display configuration information and allow the user to make the required configuration changes. The HII form is the first enabled form in the form set class EFI_HII_DRIVER_HEALTH_FORMSET_GUID, which is installed on the returned HII handle FormHiiHandle.
EfiDriverHealthStatusFailed	The controller is in a failed state and there are no actions that can place the controller into a healthy state. This controller, nor no any boot devices behind it, cannot be used as a boot device.
EfiDriverHealthStatusReconnectRequired	A hardware and/or software configuration change was performed by the user and the controller needs to be reconnected before the controller can be placed in a healthy state. The UEFI Boot Manager is required to call the UEFI Boot Service DisconnectController(), followed by the UEFI Boot Service ConnectController(), to reconnect the controller.
EfiDriverHealthStatusRebootRequired	A hardware and/or software configuration change was performed by the user and the controller requires the entire platform to be rebooted before the controller can be placed in a healthy state. The UEFI Boot Manager should complete the configuration and repair operations on all the controllers that are not in a healthy state before rebooting the system.

Depending on the specific health status value returned, additional information may be returned in *MessageList* and *FormHiiHandle* as described in the table above. The health status for devices is typically stored in the private context data structure. The Driver Binding Protocol **Start()** function for a UEFI Driver is usually where the health status for a device is initially detected and the results of that detection logic are stored in the private context data structure. As the UEFI Boot Manager performs repair or configuration actions, the health status of a controller changes. Each time **GetHealthStatus()** is called, the health status of the controller must be evaluated. The EFI Driver Health Protocol section of the *UEFI Specification* defines the legal state transitions for health status values as shown in the following figure.

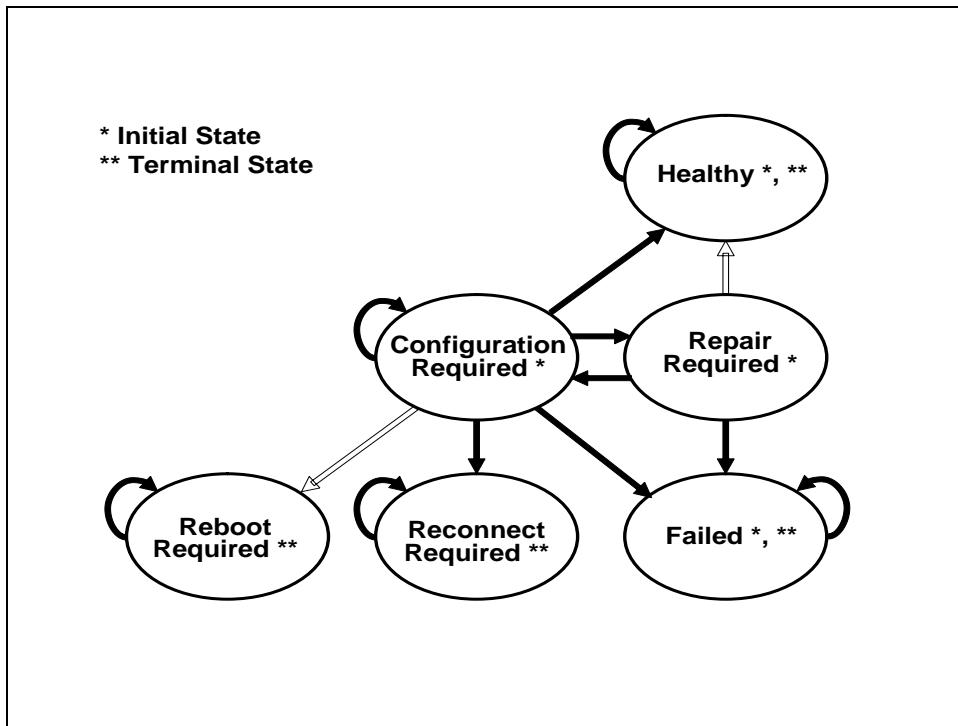


Figure 18—Driver Health Status State Diagram

The Driver Health Protocol is only available for devices a driver is currently managing. Because UEFI supports connecting the minimum number of drivers and devices required to establish console and gain access to the boot device, there may be many unconnected devices that support The Driver Health Protocol. As a result, when the user wishes to enter a platform configuration mode, the UEFI Boot Manager must connect all drivers to all devices so the UEFI Boot Manager can evaluate the health status of all the devices in the platform supporting the Driver Health Protocol.

14.2.1 Device Drivers

Device drivers that implement `GetHealthStatus()` must verify that `ChildHandle` is `NULL` and that `ControllerHandle` represents a device currently under the device driver's management. The Driver Health Protocol also supports returning the combined health status for all controllers under a UEFI Driver's management. This request is made by passing in a `ControllerHandle` value of `NULL`.

The [following example](#) shows the steps required to check these parameters. If these checks pass, the health status is returned. In this specific example, the driver opens the PCI I/O Protocol in its Driver Binding `start()` function. This is why `gEfiPciIoProtocolGuid` is used in the call to the EDK II Library `UefiLib` function `EfiTestManagedDevice()` that checks to see if the UEFI Drivers providing this `GetHealthStatus()` service is currently managing `ControllerHandle`. If the private context structure is required, typically the UEFI Boot Service `OpenProtocol()` is used to open one of the UEFI Driver produced protocols on `ControllerHandle` and then uses a `CR()` based macro to retrieve a pointer to the private context structure.

```

#include <Uefi.h>
#include <Protocol/DriverHealth.h>
#include <Protocol/PciIo.h>
#include <Library/BaseMemoryLib.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI
AbcGetHealthStatus (
    IN  EFI_DRIVER_HEALTH_PROTOCOL           *This,
    IN  EFI_HANDLE                           ControllerHandle OPTIONAL,
    IN  EFI_HANDLE                           ChildHandle   OPTIONAL,
    OUT EFI_DRIVER_HEALTH_STATUS            *HealthStatus,
    OUT EFI_DRIVER_HEALTH_HII_MESSAGE      **MessageList  OPTIONAL,
    OUT EFI_HII_HANDLE                      *FormHiiHandle OPTIONAL
)
{
    EFI_STATUS Status;

    //
    // Check input parameters
    //
    if (HealthStatus == NULL) {
        return EFI_INVALID_PARAMETER;
    }

    if (ControllerHandle == NULL) {
        //
        // If all controllers managed by this UEFI Driver are healthily,
        // then assign HealthStatus to EfiDriverHealthStatusHealthy.
        // Otherwise, assign HealthStatus to EfiDriverHealthStatusFailed.
        //
        return EFI_SUCCESS;
    }

    //
    // ChildHandle must be NULL for a Device Driver
    //
    if (ChildHandle != NULL) {
        return EFI_UNSUPPORTED;
    }

    //
    // Make sure this driver is currently managing ControllerHandle
    //
    Status = EfiTestManagedDevice (
        ControllerHandle,
        gAbcDriverBinding.DriverBindingHandle,
        &gEfiPciIoProtocolGuid
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    //
    // Retrieve health status for ControllerHandle
    //

    return EFI_SUCCESS;
}

```

Example 153—GetHealthStatus() for a Device Driver

14.2.2 Bus Drivers and Hybrid Drivers

Bus drivers and hybrid drivers implementing the Driver Health Protocol must verify that `ControllerHandle` and `ChildHandle` represent a device that is currently under the driver's management. The Driver Health Protocol also supports returning the combined health status for all controllers a UEFI Driver manages. This request is made by passing in a `ControllerHandle` value of `NULL`.

The example below shows the steps required to check these parameters and also retrieve the private context data structure. If these checks pass, the health status is returned. In this specific example, the driver opens the PCI I/O Protocol in its Driver Binding Start() function. This is why `gEfiPciIoProtocolGuid` is used in the call to the EDK II Library `UefiLib` function `EfiTestManagedDevice()` that checks to see if the UEFI Drivers providing this `GetHealthStatus()` service is currently managing `ControllerHandle`. If the private context structure is required, the UEFI Boot Service `OpenProtocol()` is typically used to open one of the UEFI Driver produced protocols on `ControllerHandle` and then uses a `CR()` based macro to retrieve a pointer to the private context structure. If diagnostics are being run on `ChildHandle`, a produced protocol on `ChildHandle` can be opened.

Note: *If `ChildHandle` is `NULL`, a request is being made to run diagnostics on the bus controller. If `ChildHandle` is not `NULL`, then a request is being made to run diagnostics on a UEFI Driver managed child controller.*

```
#include <Uefi.h>
#include <Protocol/DriverHealth.h>
#include <Protocol/PciIo.h>
#include <Library/BaseMemoryLib.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI
AbcGetHealthStatus (
    IN  EFI_DRIVER_HEALTH_PROTOCOL           *This,
    IN  EFI_HANDLE                           ControllerHandle OPTIONAL,
    IN  EFI_HANDLE                           ChildHandle   OPTIONAL,
    OUT EFI_DRIVER_HEALTH_STATUS            *HealthStatus,
    OUT EFI_DRIVER_HEALTH_HII_MESSAGE      **MessageList  OPTIONAL,
    OUT EFI_HII_HANDLE                      *FormHiiHandle OPTIONAL
)
{
    EFI_STATUS Status;

    //
    // Check input parameters
    //
    if (HealthStatus == NULL) {
        return EFI_INVALID_PARAMETER;
    }

    if (ControllerHandle == NULL) {
        //
        // If all controllers managed by this UEFI Driver are healthy,
        // then assign HealthStatus to EfiDriverHealthStatusHealthy.
        // Otherwise, assign HealthStatus to EfiDriverHealthStatusFailed.
        //
        return EFI_SUCCESS;
    }

    //

```

```

// Make sure this driver is currently managing ControllerHandle
//
Status = EfiTestManagedDevice (
    ControllerHandle,
    gAbcDriverBinding.DriverBindingHandle,
    &gEfiPciIoProtocolGuid
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// If ChildHandle is not NULL, then make sure this driver produced ChildHandle
//
if (ChildHandle != NULL) {
    Status = EfiTestChildHandle (
        ControllerHandle,
        ChildHandle,
        &gEfiPciIoProtocolGuid
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }
}

if (ChildHandle == NULL) {
    //
    // Retrieve health status for ControllerHandle
    //
} else {
    //
    // Retrieve health status for ChildHandle
    //
}

return EFI_SUCCESS;
}

```

Example 154—GetHealthStatus() for a Bus Driver or Hybrid Driver

Bus drivers and hybrid drivers are **recommended** to provide health status for both the bus controller and the child controllers these types of drivers produce. Implementing diagnostics for only the bus controller or only the child controllers is strongly discouraged.

14.3 Repair() Implementation

The **Repair()** service attempts repair operations on a driver-managed controller or a child the driver has produced. This service is not allowed to use any of the console-I/O-related protocols. Instead, the status of the repair operation is returned to the caller. The **Repair()** service supports an optional parameter called *ProgressNotification* that may be **NULL**. The caller may pass in a notification function to **Repair()** so the caller can inform the user of the progress during extended repair operations. If a repair operation takes a short period of time to execute, *ProgressNotification* may be ignored. If the repair operation takes an extended period of time to execute, the UEFI Driver should periodically call the function specified by *ProgressNotification* with *Value* and *Limit* parameters expressing the amount of repair work currently completed. The caller may choose to log or display the progress of the repair operation as well as the final results of the repair operation.

14.3.1 Device Drivers

Device drivers implementing `Repair()` must verify that `ChildHandle` is `NULL` and that `ControllerHandle` represents a device the device driver is currently managing. The following example shows the steps required to check these parameters.

If these checks pass, the health status is returned. In this specific example, the driver opens the PCI I/O Protocol in its Driver Binding `Start()` function. This is why `gEfiPciIoProtocolGuid` is used in the call to the EDK II Library `UefiLib` function `EfiTestManagedDevice()` that checks to see if the UEFI Drivers providing the `GetHealthStatus()` service is currently managing `ControllerHandle`. If the private context structure is required, typically, the UEFI Boot Service `OpenProtocol()` opens one of the UEFI Driver produced protocols on `ControllerHandle` and then uses a `CR()` based macro to retrieve a pointer to the private context structure. This example also calls `ProgressNotification` from 10% to 100% at 10% increments.

```
#include <Uefi.h>
#include <Protocol/DriverHealth.h>

EFI_STATUS
EFIAPI
AbcRepair (
    IN  EFI_DRIVER_HEALTH_PROTOCOL           *This,
    IN  EFI_HANDLE                           ControllerHandle,
    IN  EFI_HANDLE                           ChildHandle      OPTIONAL,
    IN  EFI_DRIVER_HEALTH_REPAIR_PROGRESS_NOTIFY ProgressNotification OPTIONAL
)
{
    EFI_STATUS  Status;
    UINTN       Index;

    //
    // ChildHandle must be NULL for a Device Driver
    //
    if (ChildHandle != NULL) {
        return EFI_UNSUPPORTED;
    }

    //
    // Make sure this driver is currently managing ControllerHandle
    //
    Status = EfiTestManagedDevice (
        ControllerHandle,
        gAbcDriverBinding.DriverBindingHandle,
        &gEfiPciIoProtocolGuid
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    //
    // Repair ControllerHandle
    //
    for (Index = 0; Index < 10; Index++) {
        //
        // Perform 10% of the work required to repair ControllerHandle
        //

        if (ProgressNotification != NULL) {
            ProgressNotification (Index, 10);
        }
    }
}
```

```

    }

    return EFI_SUCCESS;
}

```

Example 155—Repair() Function for a Device Driver

14.3.2 Bus Drivers and Hybrid Drivers

Bus drivers and hybrid drivers that implement the Driver Health Protocol must verify that `ControllerHandle` and `ChildHandle` represent a device that the driver is currently managing. The example below shows the steps required to check these parameters and also retrieve the private context data structure. If these checks pass, the health status is returned. In this specific example, the driver opens the PCI I/O Protocol in its Driver Binding `Start()` function.

This is why `gEfiPciIoProtocolGuid` is used in the call to the EDK II Library `UefiLib` function `EfiTestManagedDevice()` that checks to see if the UEFI Drivers providing this `Repair()` service are currently managing `ControllerHandle`. If the private context structure is required, typically the UEFI Boot Service `OpenProtocol()` is used to open one of the UEFI Driver produced protocols on `ControllerHandle` and then uses a `CR()` based macro to retrieve a pointer to the private context structure.

If diagnostics are being run on `ChildHandle`, a produced protocol on `ChildHandle` can be opened. This example also calls `ProgressNotification` from 10% to 100% at 10% increments for the bus controller and from 1% to 100%, at 1% increments, for the child controller.

Note: *If `ChildHandle` is `NULL`, a request is made to run diagnostics on the bus controller. If `ChildHandle` is not `NULL`, a request is made to run diagnostics on a UEFI Driver managed child controller.*

```

#include <Uefi.h>
#include <Protocol/DriverHealth.h>
#include <Protocol/PciIo.h>
#include <Library/BaseMemoryLib.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI
AbcRepair (
    IN  EFI_DRIVER_HEALTH_PROTOCOL           *This,
    IN  EFI_HANDLE                           ControllerHandle,
    IN  EFI_HANDLE                           ChildHandle     OPTIONAL,
    IN  EFI_DRIVER_HEALTH_REPAIR_PROGRESS_NOTIFY ProgressNotification OPTIONAL
)
{
    EFI_STATUS  Status;
    UINTN      Index;

    //
    // Make sure this driver is currently managing ControllerHandle
    //
    Status = EfiTestManagedDevice (
        ControllerHandle,
        gAbcDriverBinding.DriverBindingHandle,
        &gEfiPciIoProtocolGuid
    );
}

```

```

    );
if (EFI_ERROR (Status)) {
    return Status;
}

//
// If ChildHandle is not NULL, then make sure this driver produced ChildHandle
//
if (ChildHandle != NULL) {
    Status = EfiTestChildHandle (
        ControllerHandle,
        ChildHandle,
        &gEfiPciIoProtocolGuid
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }
}

if (ChildHandle == NULL) {
    //
    // Repair ControllerHandle
    //
    for (Index = 0; Index < 10; Index++) {
        //
        // Perform 10% of the work required to repair ControllerHandle
        //

        if (ProgressNotification != NULL) {
            ProgressNotification (Index, 10);
        }
    }
} else {
    //
    // Repair ChildHandle
    //
    for (Index = 0; Index < 100; Index++) {
        //
        // Perform 1% of the work required to repair ChildHandle
        //

        if (ProgressNotification != NULL) {
            ProgressNotification (Index, 100);
        }
    }
}

return EFI_SUCCESS;
}

```

Example 156—Repair() for a Bus Driver or Hybrid Driver

Driver Family Override Protocol

The Driver Family Override Protocol is an optional feature for UEFI Drivers following the UEFI Driver Model. The Driver Family Override Protocol provides a method for a UEFI Driver to opt-in to a higher priority rule for connecting drivers to controllers in the EFI Boot Service `ConnectController()`. This rule is higher priority than the Bus Specific Driver Override Protocol rule and lower priority than the Platform Driver Override Rule.

The Driver Family Override Protocol is typically produced by UEFI Drivers associated with a family of similar controllers when multiple versions of a UEFI Driver for a family of similar controllers are present in a platform at the same time and where the UEFI Driver writer requires that the UEFI Driver considered the highest version manage all controllers in that same family.

PCI Use Case: If a platform has 3 PCI SCSI adapters from the same manufacturer, and the manufacturer requires the PCI SCSI adapter that has the highest version UEFI Driver to manage all 3 PCI SCSI adapters, then the Driver Family Override Protocol is required and it provides the version value used to make the selection. If the Driver Family Override Protocol is not produced, then the Bus Specific Driver Override Protocol for PCI selects the UEFI Driver from the PCI Option ROM to the adapter to manage each adapter.

15.1 Driver Family Override Protocol Implementation

The implementation of the Driver Family Override Protocol is typically found in the file `DriverFamilyOverride.c`. [Appendix A](#) contains a template for a `DriverFamilyOverride.c` file for a UEFI Driver. The list of tasks to implement the Driver Family Override Protocol feature follow:

- Add global variable for the `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL` instance to `DriverFamilyOverride.c`.
- Implement the `GetVersion()` service of the Driver Family Override Protocol in `DriverFamilyOverride.c`.
- Install the Driver Family Override Protocol onto the same handle that the Driver Binding Protocol is installed.
- If the UEFI Driver produces multiple Driver Binding Protocols, install the Driver Family Override Protocol on the same handles as that of the Driver Binding Protocol.
- If the UEFI Driver supports the unload feature, uninstall all the Driver Family Override Protocol instances in the `Unload()` function.

The Driver Family Override Protocol contains one service called `GetVersion()` that returns version value used by the UEFI Boot Service `ConnectController()` to determine the order of Driver Binding Protocol used to start a specific controller. If the Driver Family Override Protocol is present, it is higher priority than the Bus Specific Driver Override Protocol, but lower than the Platform Driver Override Protocol. See the [Chapter 3](#) and the Protocol Handler Services section of the *UEFI Specification* for details

on how the UEFI Boot Service `ConnectController()` selects the best UEFI Driver to manage a specific controller.

For reference, the example below shows the protocol interface structure for the Driver Family Override Protocol. It is composed of a single service called `GetVersion()`.

```
typedef struct _EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL
  EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL;

///
/// When installed, the Driver Family Override Protocol produces a GUID that
/// represents a family of drivers. Drivers with the same GUID are members of
/// the same family. When drivers are connected to controllers, drivers with a
/// higher revision value in the same driver family are connected with a higher
/// priority than drivers with a lower revision value in the same driver family.
/// The UEFI Boot Service ConnectController() uses five rules to build a prioritized
/// list of drivers when a request is made to connect a driver to a controller.
/// The Driver Family Protocol rule is between the Platform Specific Driver
/// Override Protocol and above the Bus Specific Driver Override Protocol.
///
struct _EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL {
  EFI_DRIVER_FAMILY_OVERRIDE_GET_VERSION GetVersion;
};
```

Example 157—Driver Family Override Protocol

The following example declares a global variable called `gAbcDriverFamilyOverride` with the single service called `AbcGetVersion()`. The UEFI Boot Service `InstallMultipleProtocolInterfaces()` is used to install the Driver Family Override Protocol instance `gAbcDriverFamilyOverride` onto the same `ImageHandle` as which the Driver Binding Protocol instance `gAbcDriverBinding` is installed.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/DriverFamilyOverride.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>

#define ABC_VERSION 0x10

EFI_DRIVER_BINDING_PROTOCOL gAbcDriverBinding = {
  AbcSupported,
  AbcStart,
  AbcStop,
  ABC_VERSION,
  NULL,
  NULL
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL gAbcDriverFamilyOverride = {
  AbcGetVersion
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
  IN EFI_HANDLE           ImageHandle,
```

```

    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status;

    //
    // Install driver model protocol(s) on ImageHandle
    //
    Status = EfiLibInstallDriverBinding (
        ImageHandle,           // ImageHandle
        SystemTable,          // SystemTable
        &gAbcDriverBinding,   // DriverBinding
        ImageHandle           // DriverBindingHandle
    );
    ASSERT_EFI_ERROR (Status);

    //
    // Install Driver Family Override Protocol onto ImageHandle
    //
    Status = gBS->InstallMultipleProtocolInterfaces (
        &ImageHandle,
        &gEfiDriverFamilyOverrideProtocolGuid,
        &gAbcDriverFamilyOverride,
        NULL
    );
    ASSERT_EFI_ERROR (Status);

    return Status;
}

```

Example 158—Install Driver Family Override Protocol

15.2 GetVersion() Implementation

The example below shows an example implementation of the `GetVersion()` function of the Driver Family Override Protocol. This function returns a `UINT32` value and, in this case, returns a value from a define statement in the UEFI Driver. The manufacturer of a family of controllers is free to select any version value assignment as long as UEFI Drivers that are required to be used over previously released UEFI Drivers have higher values.

```

#include <Uefi.h>
#include <Protocol/DriverFamilyOverride.h>

#define ABC_DRIVER_FAMILY_OVERRIDE_VERSION 0x00050063

UINT32
EFIAPI
AbcGetVersion (
    IN EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL *This
)
{
    return ABC_DRIVER_FAMILY_OVERRIDE_VERSION;
}

```

Example 159—GetVersion() Function of the Driver Family Override Protocol

Driver Supported EFI Version Protocol

The Driver Supported EFI Version Protocol allows a UEFI Driver to specify the version of the *UEFI Specification* it follows. The version information follows the same format as the *Revision* field in the `EFI_TABLE_HEADER` of the EFI System Table. This feature is required for UEFI Drivers on PCI and other plug in cards, but is only recommended for all UEFI Drivers.

16.1 Driver Supported EFI Version Protocol Implementation

The implementation of the Driver Supported EFI Version Protocol is typically found in the `<>DriverName>.c` file for a UEFI Driver and is installed onto the `ImageHandle` in the driver entry point using the UEFI Boot Service `InstallMultipleProtocolInterfaces()`. [Appendix A](#) contains a template for the `<>DriverName>.c` file that includes the declaration of a global variable for the Driver Supported EFI Version Protocol instance. The list of tasks required to implement the Driver Support EFI Version Protocol feature are as follows:

- Add global variable for the `EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL` instance to `<>DriverName>.c`
- Set the `FirmwareVersion` field of the Driver Supported EFI Version Protocol instance in the driver entry point if the value required is different than the value assigned in the global variable declaration.
- Install the Driver Supported EFI Version Protocol instance onto the `ImageHandle` of the UEFI Driver in the driver entry point.
- If the UEFI Driver supports the unload feature, uninstall the Driver Supported EFI Version Protocol instance in the `Unload()` function.

The following example shows the protocol interface structure for the Driver Supported EFI Version Protocol for reference. It is composed of the two data fields called `Length` and `FirmwareVersion`.

```
///
/// The EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL provides a
/// mechanism for an EFI driver to publish the version of the EFI
/// specification it conforms to. This protocol must be placed on
/// the driver's image handle when the driver's entry point is
/// called.
///
typedef struct _EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL {
    ///
    /// The size, in bytes, of the entire structure. Future versions of this
    /// specification may grow the size of the structure.
    ///
    UINT32 Length;
    ///
}
```

```

/// The version of the EFI specification that this driver conforms to.
///
UINT32 FirmwareVersion;
} EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL;

```

Example 160—Driver Support EFI Version Protocol

This example declares a global variable called `gEbcDriverSupportedEfiVersion` whose `FirmwareVersion` field is assigned to `EFI_2_31_SYSTEM_TABLE_REVISION`, the value associated with the *UEFI 2.3.1 Specification*.

```

#include <Uefi.h>
#include <Protocol/DriverSupportedEfiVersion.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/DebugLib.h>

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL gAbcDriverSupportedEfiVersion =
{
    sizeof (EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL), // Length
    EFI_2_31_SYSTEM_TABLE_REVISION // FirmwareVersion
};

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    EFI_STATUS  Status;

    //
    // Install Driver Supported EFI Version Protocol onto ImageHandle
    //
    Status = gBS->InstallMultipleProtocolInterfaces (
        &ImageHandle,
        &gEfiDriverSupportedEfiVersionProtocolGuid,
        & gAbcDriverSupportedEfiVersion,
        NULL
    );
    ASSERT_EFI_ERROR (Status);

    return Status;
}

```

Example 161—Driver Supported EFI Version Protocol Feature

The EFI System Table chapter of the *UEFI Specification* defines revision values for the *EFI Specifications* and *UEFI Specifications*. The table below provides a summary of the define name available to UEFI Drivers.

Table 24—UEFI Specific Revision Values

Specification	Define Name	Value
<i>UEFI Specification Version 2.3.1</i>	<code>EFI_2_31_SYSTEM_TABLE_REVISION</code>	$((2 << 16) (31))$
<i>UEFI Specification Version 2.3</i>	<code>EFI_2_30_SYSTEM_TABLE_REVISION</code>	$((2 << 16) (30))$
<i>UEFI Specification Version 2.2</i>	<code>EFI_2_20_SYSTEM_TABLE_REVISION</code>	$((2 << 16) (20))$
<i>UEFI Specification Version 2.1</i>	<code>EFI_2_10_SYSTEM_TABLE_REVISION</code>	$((2 << 16) (10))$
<i>UEFI Specification Version 2.0</i>	<code>EFI_2_00_SYSTEM_TABLE_REVISION</code>	$((2 << 16) (00))$
<i>EFI Specification Version 1.1</i>	<code>EFI_1_10_SYSTEM_TABLE_REVISION</code>	$((1 << 16) (10))$
<i>EFI Specification Version 1.02</i>	<code>EFI_1_02_SYSTEM_TABLE_REVISION</code>	$((1 << 16) (02))$

UEFI Drivers producing the Driver Supported EFI Version Protocol typically use the style shown in the example above. However, more complex UEFI Drivers compatible with several versions of the *EFI Specification* and *UEFI Specification* must detect the UEFI capabilities of the platform firmware and adjust the behavior of the UEFI Driver to match those UEFI capabilities. In this more complex case, the UEFI Driver updates the *FirmwareVersion* field to declare the specific version of the *UEFI Specification* the UEFI Driver follows.

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 UEFI Drivers on their child devices.

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, a Bus Specific Driver Override Protocol is not produced for that PCI device.

17.1 Bus Specific Driver Override Protocol Implementation

The implementation of the Bus Specific Driver Override Protocol for a specific bus driver is typically found in the file `BusSpecificDriverOverride.c`. [Appendix A](#) contains a template for a `BusSpecificDriverOverride.c` file for a UEFI Driver. The list of tasks to implement the Bus Specific Driver Override Protocol feature are as follows:

- Add the `EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL` instance to the design of the private context data structure.
- Add private fields to the design of the private context data structure that support managing the set of driver image handles returned by `GetDriver()`.
- Update private content data structure initialization to initialize the Bus Specific Driver Override Protocol instance and the private fields used to manage the set of driver image handles returned by `GetDriver()`.
- Implement the `GetDriver()` service to return set-of-driver image handles from the private context data structure.
- Implement private worker functions to add/remove driver image handles from set-of-driver image handles maintained in the private context data structure.
- Install the Bus Specific Driver Override Protocol onto the child handle the bus driver produces in the Driver Binding `Start()` function.
- Uninstall the Bus Specific Driver Override Protocol from the child handle the bus driver produces in the Driver Binding `Stop()` function.

The Bus Specific Driver Override Protocol contains one service called `GetDriver()` that returns an ordered list of driver image handles for the UEFI drivers that were loaded from a container of UEFI driver(s). There are many ways to implement storage for the ordered list of driver image handled including an array and linked lists.

PCI Use Case: The order in which the image handles are returned by the PCI Bus Driver 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* for storing one or more images. The PCI bus driver walks the list of images in a PCI option ROM looking for UEFI drivers. If a UEFI driver is found, it is optionally decompressed using the Decompress Protocol and then loaded. The driver entry point is called using the UEFI boot services `LoadImage()` and `StartImage()`. If `LoadImage()` does not return an error, the UEFI driver must be added to the end of the list of drivers the Bus Specific Driver Override Protocol for that PCI device returns after the `GetDriver()` service is called.

The example below shows the protocol interface structure for the Bus Specific Driver Override Protocol for reference and is composed of a single service called `GetDriver()`.

```
typedef struct _EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL
  EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL;

///
/// This protocol matches one or more drivers to a controller. This protocol
/// is produced by a bus driver, and it is installed on the child handles of
/// buses that require a bus specific algorithm for matching drivers to
/// controllers.
///
struct _EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL {
  EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_GET_DRIVER  GetDriver;
};
```

Example 162—Bus Specific Driver Override Protocol

17.2 Private Context Data Structure

The following example shows a fragment of a private context data structure used to manage the child-device-related information in a bus driver producing the Bus Specific Driver Override Protocol. 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 returns for a single child device. The `HandleBufferSize` field is the number of handles allocated for the array `HandleBuffer`, and the `HandleBuffer` field is the array of driver image handles returned by the `GetDriver()` function of the Bus Specific Driver Override Protocol. The `CR()` macro provides a method to retrieve a pointer to an `ABC_PRIVATE_DATA` instance from a Bus Specific Driver Override Protocol `This` pointer. This macro is used by the `GetDriver()` function to retrieve the private context structure.

```
#define ABC_PRIVATE_DATA_SIGNATURE  SIGNATURE_32('A','B','C',' ')
typedef struct {
  UINTN                                     Signature;
  EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL  BusSpecificDriverOverride;
  UINTN                                     NumberOfHandles;
  UINTN                                     HandleBufferSize;
```

```

    EFI_HANDLE                                *HandleBuffer;
} ABC_PRIVATE_DATA;

#define ABC_PRIVATE_DATA_FROM_BUS_SPECIFIC_DRIVER_OVERRIDE_THIS(a) \
CR(a, ABC_PRIVATE_DATA, BusSpecificDriverOverride, ABC_PRIVATE_DATA_SIGNATURE)

```

Example 163—Private Context Data Structure with a Bus Specific Driver Override Protocol

This example shows how the private context data structure must be initialized by the bus driver when a child controller is discovered. This initialization is required for the examples of the `AbcGetDriver()` and `AbcAddDriver()` functions shown below to work correctly.

```

Private->Signature          = ABC_PRIVATE_DATA_SIGNATURE;
Private->BusSpecificDriverOverride.GetDriver = AbcGetDriver;
Private->NumberOfHandles    = 0;
Private->HandleBufferSize   = 0;
Private->HandleBuffer       = NULL;

```

Example 164—Private Context Data Structure Initialization

17.3 Bus Specific Driver Override Protocol Installation

The example below shows a fragment from the Driver Binding Protocol `start()` that installs the Bus Specific Driver Override Protocol instance onto a child handle produced by the bus driver.

```

#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/BusSpecificDriverOverride.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS
EFIAPI
AbcStart(
    IN EFI_DRIVER_BINDING_PROTOCOL  *This,
    IN EFI_HANDLE                  ControllerHandle,
    IN EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath OPTIONAL
)
{
    EFI_STATUS      Status;
    ABC_PRIVATE_DATA *Private;
    EFI_HANDLE      ChildHandle;

    ...

    Status = gBS->InstallMultipleProtocolInterfaces (
        ChildHandle,
        &gEfiBusSpecificDriverOverrideProtocolGuid,
        &Private->BusSpecificDriverOverride,
        NULL
    );
    ...
}

```

Example 165—Install Bus Specific Driver Override Protocol

The following example shows a fragment from the Driver Binding Protocol `Stop()` function that uninstalls the Bus Specific Driver Override Protocol instance from a child handle produced by the bus driver.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/BusSpecificDriverOverride.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS
EFIAPI
AbcStop(
    IN  EFI_DRIVER_BINDING_PROTOCOL      *This,
    IN  EFI_HANDLE                      ControllerHandle,
    IN  UINTN                           NumberOfChildren,
    IN  EFI_HANDLE                      *ChildHandleBuffer
)
{
    EFI_STATUS         Status;
    ABC_PRIVATE_DATA  *Private;
    EFI_HANDLE         ChildHandle;

    . .

    Status = gBS->UninstallMultipleProtocolInterfaces (
        ChildHandle,
        &gEfiBusSpecificDriverOverrideProtocolGuid,
        &Private->BusSpecificDriverOverride,
        NULL
    );
    . .
}
```

Example 166—Uninstall Bus Specific Driver Override Protocol

17.4 GetDriver() Implementation

The example below shows 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 using the `CR()` macro defined in Section 17.3 above. If no image handles are registered, `EFI_NOT_FOUND` is returned. If `DriverImageHandle` is a pointer to `NULL`, the first image handle from `HandleBuffer` is returned. If `DriverImageHandle` is not a pointer to `NULL`, a search is made through `HandleBuffer` to find a matching handle. If a matching handle is not found, `EFI_INVALID_PARAMETER` is returned. If a matching handle is found, the next handle in the array is returned. If the matching handle is the last handle in the array, `EFI_NOT_FOUND` is returned.

```
#include <Uefi.h>
#include <Protocol/BusSpecificDriverOverride.h>

EFI_STATUS
EFIAPI
AbcGetDriver (
    IN     EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL  *This,
    IN OUT EFI_HANDLE                                *DriverImageHandle
```

```

)
{
    UINTN             Index;
    ABC_PRIVATE_DATA *Private;

    Private = ABC_PRIVATE_DATA_FROM_BUS_SPECIFIC_DRIVER_OVERRIDE_THIS (This);

    if (Private->NumberOfHandles == 0) {
        return EFI_NOT_FOUND;
    }

    if (DriverImageHandle == NULL) {
        return EFI_INVALID_PARAMETER;
    }

    if (*DriverImageHandle == NULL) {
        *DriverImageHandle = Private->HandleBuffer[0];
        return EFI_SUCCESS;
    }

    for (Index = 0; Index < Private->NumberOfHandles; Index++) {
        if (*DriverImageHandle == Private->HandleBuffer[Index]) {
            Index++;
            if (Index < Private->NumberOfHandles) {
                *DriverImageHandle = Private->HandleBuffer[Index];
                return EFI_SUCCESS;
            } else {
                return EFI_NOT_FOUND;
            }
        }
    }

    return EFI_INVALID_PARAMETER;
}

```

Example 167—GetDriver() Function of a Bus Specific Driver Override Protocol

17.5 Adding Driver Image Handles

The example below shows an internal worker function that adds a driver image handle to the ordered list of driver image handles in the private context data structure. This function is used by the bus driver to register image handles associated with UEFI Drivers discovered on child devices (i.e. when the PCI bus driver discovered UEFI Drivers stored in PCI option ROMs). As each UEFI driver is loaded, this internal worker function is called to add the image handle of the UEFI driver to the Bus Specific Driver Override Protocol. The order that the image handles are registered with `AbcAddDriver()` is the order in which they are returned from `GetDriver()`.

If there is not enough room in the image handle array, an array with 10 additional handles is allocated. The contents of the old array are transferred to the new array and the old array is freed. The EDK II library `MemoryAllocationLib` provides the `ReallocatePool()` function, simplifying the implementations of UEFI Drivers required to manage dynamic memory. Lacking enough memory to allocate the new array, the `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.

```
#include <Uefi.h>
#include <Libarray/MemoryAllocationLib.h>
```

```
EFI_STATUS
EFIAPI
AbcAddDriver(
    IN ABC_PRIVATE_DATA *Private,
    IN EFI_HANDLE DriverImageHandle
)
{
    EFI_HANDLE *NewBuffer;

    if (Private->NumberOfHandles >= Private->HandleBufferSize) {
        NewBuffer = ReallocatePool (
            Private->HandleBufferSize * sizeof (EFI_HANDLE),
            (Private->HandleBufferSize + 10) * sizeof (EFI_HANDLE),
            Private->HandleBuffer
        );
        if (NewBuffer == NULL) {
            return EFI_OUT_OF_RESOURCES;
        }
        Private->HandleBufferSize += 10;
        Private->HandleBuffer = NewBuffer;
    }
    Private->HandleBuffer[Private->NumberOfHandles] = DriverImageHandle;
    Private->NumberOfHandles++;
    return EFI_SUCCESS;
}
```

Example 168—Adding Driver Image Handles

PCI Driver Design Guidelines

There are several categories of PCI drivers that cooperate to provide support for PCI controllers in a platform. Table 25— lists these PCI drivers.

Table 25—Classes of PCI drivers

Class of driver	Description
PCI 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 providing services for the console and boot devices required to boot an EFI-conformant operating system.

This chapter concentrates on the design and implementation of PCI drivers. PCI drivers must follow all of the general design guidelines described in [Chapter 4](#). This chapter covers guidelines that apply specifically to the management of PCI controllers.

The [following figure](#) shows an example PCI driver stack and the protocols 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 a child handle with the [`EFI_GRAPHICS_OUTPUT_PROTOCOL`](#).
- 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`](#). Although not shown in Figure 19, the [`EFI_USB_HOST_CONTROLLER_PROTOCOL`](#) would then be consumed by the USB bus driver to produce child handles for each USB device. USB drivers would then manage those child handles.

[Chapter 19](#) contains the guidelines for designing USB drivers.

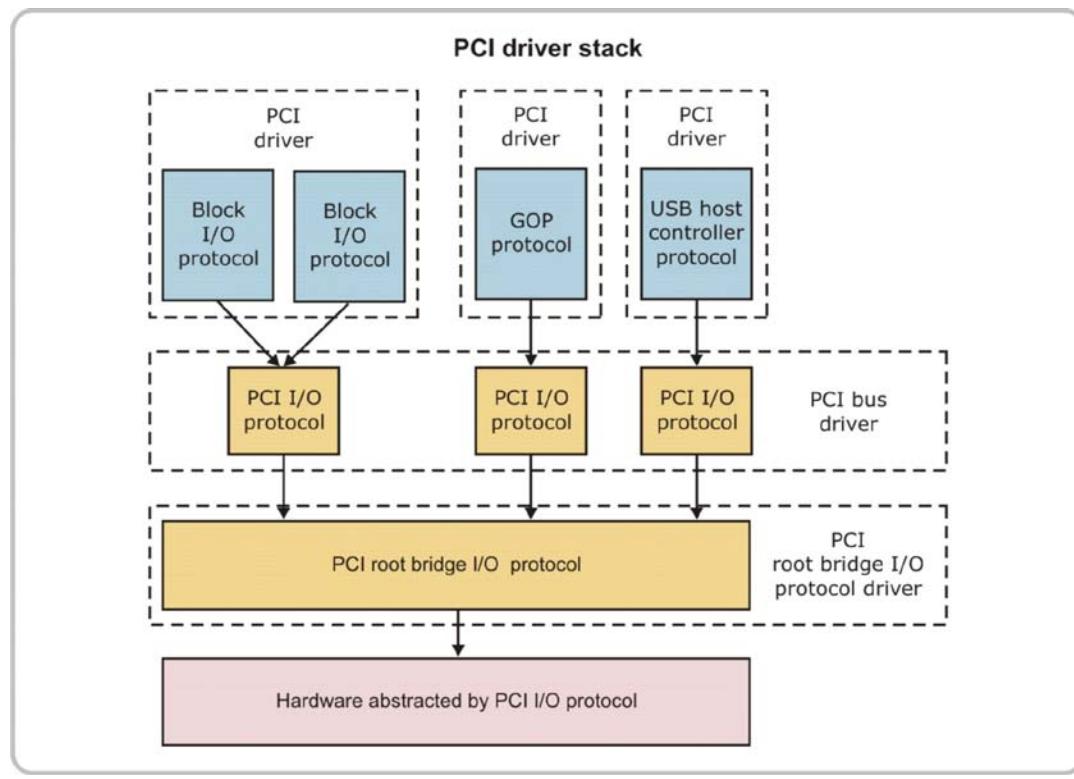


Figure 19—PCI driver stack

18.1 PCI Root Bridge I/O Protocol Drivers

UEFI firmware for a platform typically implements a Root Bridge Driver that produces the PCI Root Bridge I/O Protocol. This code is chipset specific and directly accesses the chipset resources producing the services of the PCI Root Bridge I/O Protocol. A sample driver for systems with a PC-AT-compatible chipset is included in EDK II. The source code for this driver is found in the EDK II package called [PcAtChipsetPkg](#) in the directory [PcAtChipsetPkg/PciHostBridgeDxe](#).

18.2 PCI Bus Drivers

EDK II contains a generic PCI bus driver. It uses the services of the [EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL](#) to enumerate PCI devices and produce a child handle with an [EFI_DEVICE_PATH_PROTOCOL](#) and an [EFI_PCI_IO_PROTOCOL](#). The source code to this driver is in the EDK II package called [MdeModulePkg](#) in the directory [MdeModulePkg/Bus/Pci/PciBusDxe](#).

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 for a PCI bus generally does not make sense. This is 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 enumerated PCI devices. Because of this, it is recommended that the PCI bus driver produce all of the PCI devices on the first call to `Start()`.

If a UEFI based system firmware is ported to a new platform, most of the PCI-related changes occur in the implementation of the Root Bridge Driver producing the `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` instances.

TIP: PCI Bus Driver customizations are **strongly discouraged** because the PCI Bus Driver is designed to be conformant with the *PCI Specification*. Instead, focus platform specific customizations on the Root Bridge Driver that produced `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` and its PCI Device Drivers.

18.2.1 Hot-plug PCI buses

The PCI bus driver in the EDK II does not support hot-plug events in the pre-boot environment. The PCI bus driver functions correctly with hot-plug-capable hardware, but the hot-add, hot-remove, and hot-replace events are only supported while an OS that supports hot-plug events is executing. The PCI bus driver requires updates to support hot-plug events in the pre-boot environment.

18.3 PCI drivers

PCI drivers use the services of the `EFI_PCI_IO_PROTOCOL` to produce one or more protocols providing 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

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 using the SCSI command set typically produce the `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` for each SCSI channel the disk controller produces. [Chapter 20](#) covers details on SCSI drivers.

18.3.1 Supported()

A PCI driver must implement the `EFI_DRIVER_BINDING_PROTOCOL` containing the `Supported()`, `Start()`, and `Stop()` services. The `Supported()` service evaluates the `ControllerHandle` passed in to see if the `ControllerHandle` represents a PCI device the PCI driver can manage.

The most common method of implementing the 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 the PCI driver knows how to manage, `Supported()` returns `EFI_SUCCESS`. Otherwise, the `Supported()` service returns `EFI_UNSUPPORTED`. The PCI driver must be careful not to disturb the state of the PCI controller because a different PCI driver may be managing the PCI controller.

Caution: Do not allow functions to “touch” or change the state of any hardware device in the `Supported()` function of the Driver Binding Protocol. Doing so can significantly degrade the driver’s performance and/or cause the device, the driver, and/or other drivers to lose sync and behave badly and unpredictably.

TIP: When modifying PCI device registers, be careful with the bits in the PCI device configuration space. Perform a read, then modify the desired bits, then do a write. Do not perform only a write operation to the bits, since that can reset other bits in the register.

The following example shows an example of the Driver Binding Protocol `Supported()` service for the ABC PCI driver managing a PCI controller with a vendor ID of 0x8086 and a device ID of 0xFFFF.

First, it attempts to open the PCI I/O Protocol `EFI_OPEN_PROTOCOL_BY_DRIVER` with `OpenProtocol()`. If the PCI I/O Protocol cannot be opened, the PCI driver does not support the controller specified by `ControllerHandle`. If the PCI I/O Protocol is opened, the services of the PCI I/O Protocol are used to read the vendor ID and device ID from the PCI configuration header. Always closed the PCI I/O Protocol with `CloseProtocol()`. `EFI_SUCCESS` is returned if the vendor ID and device ID match.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/PciIo.h>
#include <IndustryStandard/Pci.h>
#include <Library/UefiBootServicesTableLib.h>

#define ABC_VENDOR_ID 0x8086
#define ABC_DEVICE_ID 0xFFFFE

EFI_STATUS
EFIAPI
AbcSupported (
    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 16-bit Vendor ID from the PCI configuration header at offset 0x00
//
Status = PciIo->Pci.Read (
    PciIo,                               // This
    EfiPciIoWidthUint16,                // Width
    PCI_VENDOR_ID_OFFSET,               // Offset
    sizeof (VendorId),                 // Count
    &VendorId                          // Buffer
);
if (EFI_ERROR (Status)) {
    goto Done;
}

//
// Read 16-bit Device ID from the PCI configuration header at offset 0x02
//
Status = PciIo->Pci.Read (
    PciIo,                               // This
    EfiPciIoWidthUint16,                // Width
    PCI_DEVICE_ID_OFFSET,               // Offset
    sizeof (DeviceId),                 // Count
    &DeviceId                          // Buffer
);
if (EFI_ERROR (Status)) {
    goto Done;
}

//
// Evaluate Vendor ID and Device ID
//
Status = EFI_SUCCESS;
if (VendorId != ABC_VENDOR_ID || DeviceId != ABC_DEVICE_ID) {
    Status = EFI_UNSUPPORTED;
}

Done:
//
// Close the PCI I/O Protocol
//
gBS->CloseProtocol (
    ControllerHandle,
    &gEfiPciIoProtocolGuid,
    This->DriverBindingHandle,
    ControllerHandle
);

return Status;
}

```

Example 169—Supported() Reading 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 were read at once. However, this would increase the execution time because the `Supported()` service reads the entire PCI configuration header for every `ControllerHandle` passed in.

The `Supported()` service is intended to be a small, quick check. If a more extensive evaluation of the PCI configuration header is required, it may make sense to read the entire PCI configuration header at once. The example below shows the same example as above, but differs in that it reads the entire PCI configuration header in a single call to the PCI I/O Protocol reading, 32-bits at a time.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/PciIo.h>
#include <IndustryStandard/Pci.h>
#include <Library/UefiBootServicesTableLib.h>

#define ABC_VENDOR_ID 0x8086
#define ABC_DEVICE_ID 0xFFFFE

EFI_STATUS
EFIAPI
AbcSupported (
    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 using 32-bit reads
    //
    Status = PciIo->Pci.Read (
        PciIo,                                // This
        EfiPciIoWidthUint32,                  // Width
        0,                                     // Offset
        sizeof (Pci) / sizeof (UINT32),         // Count
        &Pci                                    // Buffer
    );
    if (EFI_ERROR (Status)) {
        goto Done;
    }

    //
    // Evaluate Vendor ID and Device ID
    //

    // Check if Vendor ID and Device ID are valid
    if (ABC_VENDOR_ID != Pci.VendorID || ABC_DEVICE_ID != Pci.DeviceID) {
        return EFI_UNSUPPORTED;
    }
}
```

```

//  

Status = EFI_SUCCESS;  

if (Pci.Hdr.VendorId != ABC_VENDOR_ID ||  

    Pci.Hdr.DeviceId != ABC_DEVICE_ID ) {  

    Status = EFI_UNSUPPORTED;  

}  

Done:  

//  

// Close the PCI I/O Protocol  

//  

gBS->CloseProtocol (   

    ControllerHandle,  

    &gEfiPciIoProtocolGuid,  

    This->DriverBindingHandle,  

    ControllerHandle  

);  

return Status;
}

```

Example 170—Supported() Reading entire PCI Configuration Header

18.3.2 Start() and Stop()

The `start()` service of the Driver Binding Protocol for a PCI driver also opens the PCI I/O Protocol with an attribute of `EFI_OPEN_PROTOCOL_BY_DRIVER`. If the PCI driver is a bus or hybrid driver, the Device Path Protocol opens using the attribute `EFI_OPEN_PROTOCOL_BY_DRIVER`. A device driver is not required to open the Device Path Protocol. 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 that of the `stop()` service to restore these bits. In order for the `stop()` service to restore the attributes, a PCI Driver typically stores the original attributes in a `UINT64` field of the private context data structure.

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, 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 assumes 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 perform PCI DMA transactions. If a PCI bus master that is capable of 32-bit DMA addresses is present in a platform supporting more than 4 GB of system memory, the DMA transactions may have to be double buffered. 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.

The example below shows the code fragment from the `Start()` services of a PCI driver for a PCI controller supporting 64-bit DMA transactions. The example opens the PCI I/O Protocol attribute of `EFI_OPEN_PROTOCOL_BY_DRIVER`. It then retrieves the current set of PCI I/O Protocol attributes and saves them in the private context data structure field called `ABC_PRIVATE_DATA`. It then determines what attribute the PCI I/O Protocol supports and enables the I/O decode, MMIO decode, and Bus Master, and Dual Address Cycle capabilities. If a PCI Controller does not support DAC, the only change is the removal of `EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE` from the last call to the `Attributes()` service of the PCI I/O Protocol.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/PciIo.h>
#include <Library/UefiBootServicesTableLib.h>

typedef struct {
    UINTN                 Signature;
    //
    // ...
    //
    UINT64                OriginalPciAttributes;
} ABC_PRIVATE_DATA;

EFI_STATUS               Status;
EFI_DRIVER_BINDING_PROTOCOL *This;
EFI_HANDLE               ControllerHandle;
ABC_PRIVATE_DATA         *Private;
EFI_PCI_IO_PROTOCOL      *PciIo;
UINT64                  PciSupports;

//
// Open the PCI I/O Protocol
//
Status = gBS->OpenProtocol (
    ControllerHandle,
    &gEfiPciIoProtocolGuid,
    (VOID **)&PciIo,
    This->DriverBindingHandle,
    ControllerHandle,
    EFI_OPEN_PROTOCOL_BY_DRIVER
);
if (EFI_ERROR (Status)) {
    goto Done;
}

//
// Retrieve original PCI attributes and save them in the private context data
// structure.
//
Status = PciIo->Attributes (
    PciIo,
    EfiPciIoAttributeOperationGet,
    0,
    &Private->OriginalPciAttributes
);
if (EFI_ERROR (Status)) {
    goto Done;
}

//
// Retrieve attributes that the PCI Controller supports
//
Status = PciIo->Attributes (
```

```

        PciIo,
        EfiPciIoAttributeOperationSupported,
        0,
        &PciSupports
    );
if (EFI_ERROR (Status)) {
    goto Done;
}

//
// Enable Command register and Dual Address Cycle
//
Status = PciIo->Attributes (
    PciIo,
    EfiPciIoAttributeOperationEnable,
    (PciSupports & EFI_PCI_DEVICE_ENABLE) |
    EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE,
    NULL
);
if (EFI_ERROR (Status)) {
    goto Done;
}

```

Example 171—Start() for a 64-bit DMA-capable PCI controller

This example shows the code fragment from the `stop()` services of a PCI driver. This example restores the PCI I/O Protocol attributes from a field of the private context data structure called `ABC_PRIVATE_DATA`.

```

#include <Uefi.h>
#include <Protocol/PciIo.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS           Status;
EFI_PCI_IO_PROTOCOL *PciIo;
ABC_PRIVATE_DATA     *Private;

//
// Restore original PCI attributes
//
PciIo->Attributes (
    PciIo,
    EfiPciIoAttributeOperationSet,
    Private->OriginalPciAttributes,
    NULL
);

//
// Close the PCI I/O Protocol
//
gBS->CloseProtocol (
    ControllerHandle,
    &gEfiPciIoProtocolGuid,
    This->DriverBindingHandle,
    ControllerHandle
);

```

Example 172—Restore PCI Attributes in Stop()

The following table lists the `#define` statements compatible 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 are forwarded to the device. The call to `Attributes()` fails if the request cannot be satisfied. If this failure occurs, the `Start()` function must return an error.

Once again, any attributes enabled in the `start()` service must be restored in the `Stop()` service.

Table 26—PCI Attributes

Attribute	Description
<code>EFI_PCI_IO_ATTRIBUTE_ISA_MOTHERBOARD_IO</code>	Used to request the forwarding of I/O cycles 0x0000–0x00FF (10-bit decode).
<code>EFI_PCI_IO_ATTRIBUTE_ISA_IO</code>	Used to request the forwarding of I/O cycles 0x100–0x3FF (10-bit decode).
<code>EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO</code>	Used to request the forwarding of I/O cycles 0x3C6, 0x3C8, and 0x3C9 (10-bit decode).
<code>EFI_PCI_IO_ATTRIBUTE_VGA_MEMORY</code>	Used to request the forwarding of MMIO cycles 0xA0000–0xBFFFF (24-bit decode).
<code>EFI_PCI_IO_ATTRIBUTE_VGA_IO</code>	Used to request the forwarding of I/O cycles 0x3B0–0x3BB and 0x3C0–0x3DF (10-bit decode).
<code>EFI_PCI_IO_ATTRIBUTE_IDE_PRIMARY_IO</code>	Used to request the forwarding of I/O cycles 0x1F0–0x1F7, 0x3F6, 0x3F7 (10-bit decode).
<code>EFI_PCI_IO_ATTRIBUTE_IDE_SECONDARY_IO</code>	Used to request the forwarding of I/O cycles 0x170–0x177, 0x376, 0x377 (10-bit decode).
<code>EFI_PCI_IO_ATTRIBUTE_IO</code>	Enable the I/O decode bit in the Command register.
<code>EFI_PCI_IO_ATTRIBUTE_MEMORY</code>	Enable the Memory decode bit in the Command register.
<code>EFI_PCI_IO_ATTRIBUTE_BUS_MASTER</code>	Enable the Bus Master bit in the Command register.
<code>EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE</code>	Clear for PCI controllers that cannot generate a DAC.
<code>EFI_PCI_IO_ATTRIBUTE_ISA_IO_16</code>	Used to request the forwarding of I/O cycles 0x100–0x3FF (16-bit decode).
<code>EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO_16</code>	Used to request the forwarding of I/O cycles 0x3C6, 0x3C8, and 0x3C9 (16-bit decode).
<code>EFI_PCI_IO_ATTRIBUTE_VGA_IO_16</code>	Used to request the forwarding of I/O cycles 0x3B0–0x3BB and 0x3C0–0x3DF (16-bit decode).

The table below, lists `#define` statements *not* part of the *UEFI Specification*, but which are included in EDK II to simplify PCI driver implementations. These attributes cover

the typical classes of hardware capabilities and provide a names for common combinations of attributes described in the PCI Bus Support chapter of the *UEFI Specification*.

TIP: For code readability, the Enable attributes included in EDK II should be used.

Table 27—EDK II attributes #defines

Attribute	Description
<code>EFI_PCI_DEVICE_ENABLE</code>	Equivalent to a logical OR combination of <code>EFI_PCI_IO_ATTRIBUTE_IO</code> , <code>EFI_PCI_IO_ATTRIBUTE_MEMORY</code> , and <code>EFI_PCI_IO_ATTRIBUTE_BUS_MASTER</code> .
<code>EFI_VGA_DEVICE_ENABLE</code>	Equivalent to a logical OR combination of <code>EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO</code> , <code>EFI_PCI_IO_ATTRIBUTE_VGA_MEMORY</code> , <code>EFI_PCI_IO_ATTRIBUTE_VGA_IO</code> , and <code>EFI_PCI_IO_ATTRIBUTE_IO</code> .

This table lists the `#define` statements that to use with the `GetBarAttributes()` and `SetBarAttributes()` services to adjust the attributes of a memory-mapped I/O region described by a Base Address Register (BAR) of a PCI controller. The support of these attributes is optional, but in general, a PCI driver uses these attributes to provide hints that may be used to improve the performance of a PCI driver. Improved performance is especially important for PCI drivers managing graphics controllers. Do note that any BAR attributes set in the `start()` service must be restored in the `stop()` service.

Table 28—PCI BAR attributes

Attribute	Description
<code>EFI_PCI_IO_ATTRIBUTE_MEMORY_WRITE_COMBINE</code>	Setting this bit enables platform support for memory range access in a write-combining mode. It improves write performance to a memory buffer on a PCI controller. By default, PCI memory ranges are not accessed in a write combining mode.
<code>EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED</code>	Setting this bit enables platform support for changing the attributes of a PCI memory range so that it is accessed in a cached mode. By default, PCI memory ranges are not cached.
<code>EFI_PCI_IO_ATTRIBUTE_MEMORY_DISABLE</code>	Setting this bit enables platform support for disabling a PCI memory range so that it can no longer be accessed. By default, all PCI memory ranges are enabled.

Sometimes there may be different logic paths in a UEFI Driver between a PCI add-in card and a PCI controller integrated into a platform. The PCI I/O Protocol provides attributes that help a UEFI Driver determine if a specific PCI Controller and its associated PCI Option ROM image are from a PCI add-in card in a PCI slot or if they are

integrated into a platform. The attributes shown in the following table list the `#define` statements for these attributes. These attributes are read-only and the values are established by the PCI Bus Driver when a PCI Controller is discovered and the PCI I/O Protocol is produced. A PCI driver may retrieve the attributes of a PCI controller with the `Attributes()` service of the PCI I/O Protocol, but a PCI Driver is not allowed to modify these attributes.

Table 29—PCI Embedded Device Attributes

Attribute	Description
<code>EFI_PCI_IO_ATTRIBUTE_EMBEDDED_DEVICE</code>	If this bit is set, the PCI controller is an embedded device; typically a component on the system board. If this bit is clear, the PCI controller is part of an adapter populating one of the systems PCI slots.
<code>EFI_PCI_IO_ATTRIBUTE_EMBEDDED_ROM</code>	If this bit is set, the PCI option ROM described by the <code>RomImage</code> and <code>RomSize</code> fields is not from ROM BAR of the PCI controller. If this bit is clear, the <code>RomImage</code> and <code>RomSize</code> fields were initialized based on the PCI option ROM found through the ROM BAR of the PCI controller.

18.3.3 PCI Cards with Multiple PCI 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 if they need to be controlled by a single instance of the UEFI driver. Take the following figure as a sample device.

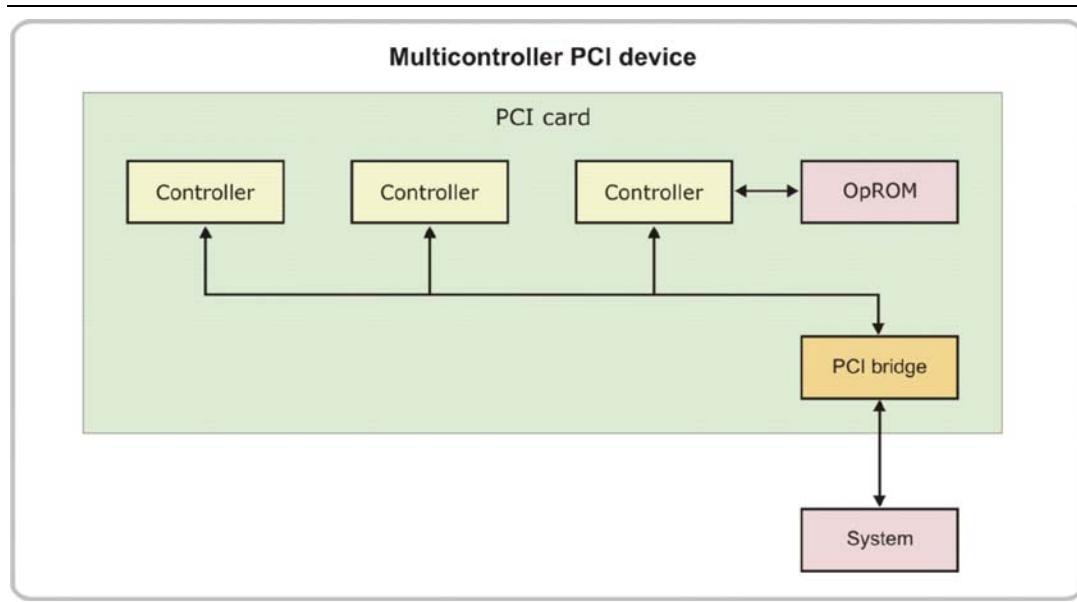


Figure 20—A multi-controller PCI device

It may be required that the driver in the Option ROM control all 3 controllers on the PCI device. To do this, use the following actions:

- In the `Supported()` function, make sure that the UEFI Driver supports the controller that is passed into the `Support()` function. The first controller passed in could be any of the controllers on a PCI Card.
- In the `Supported()` function, make sure the function does not touch or change the HW state. *This is very important.* If the PCI I/O instance is already opened (if some other application or driver is already managing the controller), return an error. See [Chapter 9](#) of this guide and the EFI Driver Binding Protocol section of the *UEFI Specification* for details on the error return codes from the Driver Binding Protocol `Supported()` function.
- In the `Start()` function for the first controller on the PCI Card, open the PCI I/O protocol instances on the other handles on the same PCI Card using the `EFI_OPEN_PROTOCOL_BY_DRIVER` attribute. This informs all other UEFI Drivers in the platform that all the controllers on the PCI Card are already being managed.
 - To scan for other PCI controllers on the same PCI Card, use the UEFI Boot Service `LocateHandleBuffer()` to find all handles in the Handle Database supporting the PCI I/O Protocol. Use the PCI I/O Protocol function `GetLocation()` to evaluate whether or not the PCI controller is on the same PCI bus number. Take care to not modify the HW state of any PCI I/O Protocol instance during this evaluation.
- In the `Stop()` function, undo everything that was done in `Start()`. Use a private context data structure to keep track of the information required to complete a `stop()` operation in these more complex use cases.

18.4 Accessing PCI resources

PCI drivers should only access the I/O and memory-mapped I/O resources on the PCI controllers 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 easily access the resources of the PCI controllers it is currently managing. These services hide platform-specific implementation details and prevent a PCI driver from inadvertently accessing 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 allowing a PCI driver to access the resources of a PCI controller include the following:

- `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 provided through the PCI I/O Protocol is the contents of the PCI option ROM. 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.

Note: *It is important that the PCI option ROM contents not be modified through the `RomImage` field. Modifications to this buffer only modify the copy of the PCI option ROM contents in system memory. The PCI I/O Protocol does not provide services to modify the content of the actual PCI option ROM.*

18.4.1 Memory-mapped I/O ordering issues

PCI transactions follow the ordering rules defined in the *PCI 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 guaranteed to complete before control is returned to the PCI driver. However, the PCI I/O Protocol `Mem.Write()` service does not guarantee that PCI memory cycles produced by this service are completed 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 to this issue than others.

The following example 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 guarantees that the write transaction is completed by the time the `Mem.Read()` call returns.

In general, this mechanism is not required because a PCI driver typically reads a status register and this read transaction forces all posted write transactions to complete on the PCI controller. The only time to use this mechanism is when a PCI driver performs a write transaction not immediately followed by a read transaction and the PCI driver needs to guarantee that the write transaction is completed immediately.

```
#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT32 DmaStartAddress;

//
// Write the value in DmaStartAddress to offset 0x20 of BAR #1
//
Status = PciIo->Mem.Write (
```

```

PciIo,           // This
EfiPciIoWidthUint32, // Width
1,               // BarIndex
0x20,            // Offset
1,               // Count
&DmaStartAddress // Buffer
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Read offset 0x20 of BAR #1. This guarantees that the previous write
// transaction is posted to the PCI controller.
//
Status = PciIo->Mem.Read (
    PciIo,           // This
    EfiPciIoWidthUint32, // Width
    1,               // BarIndex
    0x20,            // Offset
    1,               // Count
    &DmaStartAddress // Buffer
);
if (EFI_ERROR (Status)) {
    return Status;
}

```

Example 173—Completing a memory write transaction

18.4.2 Hardfail/Softfail

PCI drivers must make sure they do not access resources 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, that the PCI controller does respond. In general, this is not a concern because the PCI I/O Protocol services 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 providing 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 desired PCI controller does respond to the I/O or memory-mapped I/O requests in the ISA ranges. The PCI driver can typically know if it responds 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. The following is an example using the `EFI_PCI_IO_PASS_THROUGH_BAR`.

```
#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_PROTOCOL *PciIo;
UINT8 Data;
UINT16 Word;

//
// Write 0xAA to a Post Card at ISA address 0x80
//
Data = 0xAA;
Status = PciIo->Io.Write(
    PciIo,
    EfiPciIoWidthUint8,
    EFI_PCI_IO_PASS_THROUGH_BAR, // BarIndex
    0x80, // Offset
    1, // Count
    &Data // Buffer
);
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Read the first word from the VGA frame buffer
//
Status = PciIo->Mem.Read(
    PciIo,
    EfiPciIoWidthUint16,
    EFI_PCI_IO_PASS_THROUGH_BAR, // BarIndex
    0xA0000, // Offset
    1, // Count
    &Word // Buffer
);
if (EFI_ERROR (Status)) {
    return Status;
}
```

Example 174—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. It can retrieve the parent PCI Root Bridge I/O Protocol and use those services to touch any resource on the PCI bus.

Caution: *This touching of resources on the PCI bus 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 is best left to OEM drivers having intimate knowledge of the platform and chipset.*

[Chapter 5](#) discusses the use of the `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 needing access to the resources of other PCI controllers in the platform should search the Handle Database for controller handles supporting the PCI I/O Protocol. To prevent resource

conflicts, open PCI I/O Protocols from other PCI controllers with `EFI_OPEN_PROTOCOL_BY_DRIVER`.

The following example 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 peer PCI controllers.

For example, a PCI adapter containing 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 connects to all the other PCI controllers having the same bus number as the first. This example takes advantage of the `GetLocation()` service of the PCI I/O Protocol to find matching bus numbers.

```
#include <Uefi.h>
#include <Protocol/PciIo.h>
#include <Library/MemoryAllocationLib.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINTN HandleCount;
EFI_HANDLE *HandleBuffer;
UINTN Index;
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 that support the PCI I/O Protocol,
// and retrieve the instance of the PCI I/O Protocol. Use the
// EFI_OPEN_PROTOCOL_BY_DRIVER open mode, so only PCI I/O Protocols
// that are not currently being managed are considered.
//
for (Index = 0; Index < HandleCount; Index++) {
    Status = gBS->OpenProtocol (
        HandleBuffer[Index],
        
```

```

        &gEfiPciIoProtocolGuid,
        (VOID **)&PciIo,
        gImageHandle,
        NULL,
        EFI_OPEN_PROTOCOL_BY_DRIVER
    );
if (EFI_ERROR (Status)) {
    continue;
}

//
// 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) && Bus != MyBus) {
    //
    // 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,
        gImageHandle,
        NULL
    );
    continue;
}

//
// 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
//
}

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

```

Example 175—Locate PCI handles with matching bus number

18.4.3 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 does not create a child handle for a PCI controller that does not have any allocated resources, and as a result, a PCI driver is never passed a *ControllerHandle* for a PCI controller not having 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 never has to handle this case.

18.5 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 used to manage PCI DMA transactions include:

- `PciIo->AllocateBuffer()`
- `PciIo->FreeBuffer()`
- `PciIo->Map()`
- `PciIo->Unmap()`
- `PciIo->Flush()`

18.5.1 Map() Service Cautions

A common mistake in writing PCI drivers is omission of the use of the `Map()` service. On platforms with coherent PCI busses having a 1:1 mapping between CPU addresses and PCI DMA addresses, such as PCI implementations on many IA32, X64, and IPF systems, the omission of `Map()` may not produce any functional issues. However, if those same UEFI Driver sources are used on a platform is that not coherent, nor guarantees a 1:1 mapping between CPU addresses and PCI DMA addresses, the UEFI Driver may not function correctly, with the likely result being data corruption. For this reason, `Map()` must always be used when setting up a PCI DMA transfer.

TIP: Although omission of the `Map()` service may work on some platforms, use of `Map()` for DMA transaction is required and maximizes UEFI Driver compatibility.

The `Map()` service converts a system memory address to an address useful to a PCI device performing bus master DMA transactions. The device address 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 are the same. However, a PCI driver cannot tell if it is 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 the PCI driver is managing.

The [example 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 must never use the returned parameter `DeviceAddress` to access the contents of the DMA buffer.

```

/***
 * Provides the PCI controller-specific addresses needed to access system memory.
 *
 * @param This
 *          A pointer to the EFI_PCI_IO_PROTOCOL instance.
 * @param Operation
 *          Indicates if the bus master is going to read or write to system memory.
 * @param HostAddress
 *          The system memory address to map to the PCI controller.
 * @param NumberOfBytes
 *          On input the number of bytes to map. On output the number of bytes that were mapped.
 * @param DeviceAddress
 *          The resulting map address for the bus master PCI controller to use to access the hosts HostAddress.
 * @param Mapping
 *          A resulting value to pass to Unmap().
 *
 * @retval EFI_SUCCESS
 *          The range was mapped for the returned NumberOfBytes.
 * @retval EFI_UNSUPPORTED
 *          The HostAddress cannot be mapped as a common buffer.
 * @retval EFI_INVALID_PARAMETER
 *          One or more parameters are invalid.
 * @retval EFI_OUT_OF_RESOURCES
 *          The request could not be completed due to a lack of resources.
 * @retval EFI_DEVICE_ERROR
 *          The system hardware could not map the requested address.
 *
 */
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_MAP)(
    IN EFI_PCI_IO_PROTOCOL           *This,
    IN EFI_PCI_IO_PROTOCOL_OPERATION Operation,
    IN VOID                          *HostAddress,
    IN OUT UINTN                     *NumberOfBytes,
    OUT EFI_PHYSICAL_ADDRESS         *DeviceAddress,
    OUT VOID                         **Mapping
);

```

Example 176—Map() Function

18.5.2 Weakly ordered memory transactions

Some processors, such as those in IPF platforms, have weakly ordered memory models. With weak ordering, 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 on the use of the EDK II library **BaseLib** function **MemoryFence()** (see the next discussion) should be followed even if the driver is not initially implemented for an IPF platform. The techniques shown here do not have any impact on the executable size of a driver for strongly ordered processors such as IA32, X64, and EBC.

18.5.3 Bus Master Read and Write Operations

When a DMA transaction starts or stops, the ownership of the DMA buffer transitions from the processor to the DMA bus master and back to the processor. The PCI I/O Protocol provides the **Map()** and **Unmap()** services used to set up and complete a DMA transaction.

The implementation of the PCI Root Bridge I/O Protocol uses the EDK II library **BaseLib** function **MemoryFence()** to guarantee all system memory transactions from the processor are completed before the DMA transaction is started. This prevents a DMA

bus master reading from a location in the DMA buffer before a write transaction is flushed from the processor. Because this functionality is built into the PCI Root Bridge I/O Protocol itself, the PCI driver writer need not worry about bus master read/ 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 posted write operations are flushed from the PCI controller and through any PCI-to-PCI bridges between the PCI controller and the PCI root bridge.

Because PCI drivers are polled, they 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 and guarantees their commitment 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.

An example of how a bus master write transaction should be completed to guarantee the bus master's view of system memory is consistent with that of the processor follows.

```
#include <Uefi.h>
#include <Protocol/PciIo.h>
#include <Library/UefiLib.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT64 Result64

//
// Call PollMem() to poll for Bit #0 in MMIO register 0x24 of Bar #1 to be set.
// This example shows polling a status register to wait for a bus master write
// transaction to complete.
//
Status = PciIo->PollMem (
    PciIo,                               // This
    EfiPciIoWidthUint32,                 // Width
    1,                                    // BarIndex
    0x24,                                // Offset
    BIT0,                                 // Mask
    BIT0,                                 // Value
    EFI_TIMER_PERIOD_SECONDS (1),        // Timeout
    &Result64                             // Result
);
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;
}
```

Example 177—Completing a bus master write operation

18.5.4 Bus Master Common Buffer Operations

Bus master common buffer operations are more complex to manage than bus master read and write operations, 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, no hazards are present. If the processor has deep write buffers, a write transaction may be delayed. The EDK II library `BaseLib` provides the `MemoryFence()` function to force completion of all processor transactions. If a memory region to which the processor needs to read or write requires multiple atomic processor transactions, hazards may exist if the operations are reordered. If the order in which the processor transactions occur is important, insert the `MemoryFence()` between the processor transactions. Use sparingly, though. Inserting too many `MemoryFence()` calls may degrade system performance. For strongly ordered processors, the `MemoryFence()` function is a no-op.

A good example of `MemoryFence()` use is that of a mailbox data structure 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 scans the mailbox to see if the valid bit is set. When the bus master sees the valid bit is set, it reads the rest of the mailbox contents and uses them to perform an I/O operation. If the processor is weakly ordered, there is a chance that the valid bit is set before the processor has written all of the other fields in the data structure. To resolve this issue, a `MemoryFence()` call is inserted just before and just after the valid bit is set.

Another mechanism used to resolve these memory-ordering issues is that of the `volatile` keyword in C sources. If the data structure used as a mailbox is declared in C as `volatile`, the C compiler guarantees that all transactions to the `volatile` data structure are strongly ordered. It is recommended that the `MemoryFence()` call be used instead of `volatile` data structures.

18.5.5 4 GB Memory Boundary

32-bit platforms may support more than 4 GB of system memory, but UEFI drivers for 32-bit platforms may only access memory below 4 GB. The 4 GB memory boundary becomes more complex on 64-bit platforms. Also, some 64-bit platforms may not map any system memory in the memory region below 4 GB. For more information about the 4 GB memory boundary on various architectures, see [Section 4.2](#) of this guide.

A UEFI driver should not allocate buffers from, 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, or below, a specific address. Also, UEFI drivers should always use the services of the PCI I/O Protocol to set up and complete DMA transactions.

Caution: *It is not legal to program a system memory address into a DMA bus master. Such programming may function correctly on platforms having a one-to-one mapping between system memory addresses and PCI DMA addresses, but it will not work on*

platforms that remap DMA transactions, nor on platforms using a virtual addressing mode for system memory addresses not one-to-one mapped to the PCI DMA addresses.

The following sections contain code examples for the different types of PCI DMA transactions supported by the *UEFI Specification*. It shows how to best use the PCI I/O Protocol services to maximize the platform compatibility of UEFI drivers.

EDK II contains an implementation of the PCI Root Bridge I/O Protocol for a PC-AT-compatible chipset, and assumes a one-to-one mapping between system memory 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 that buffer below 4 GB.

Note: *It is important to realize that these functions are implemented differently for platforms not assuming a one-to-one mapping between system memory addresses and PCI DMA addresses or if the platform can only perform DMA in specific ranges of system memory.*

18.5.6 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 [following example](#) 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 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 the function uses to check 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 used in this example include `Map()`, `Unmap()`, `Mem.Write()`, and `PollMem()`. The example below is for a 32-bit PCI bus master.

A 64-bit PCI bus master instance uses 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, the `EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE` attribute must be set in the Driver Binding Protocol `Start()` service of the PCI driver.

```

#include <Uefi.h>
#include <Protocol/PciIo.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI
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;
    UINT64              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,                               // This
            EfiPciIoOperationBusMasterRead,       // Operation
            (VOID *)HostAddress,                 // HostAddress
            &NumberOfBytes,                     // NumberOfBytes
            &DeviceAddress,                    // DeviceAddress
            &Mapping,                          // 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,                               // This
            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 also starts the DMA transaction
        //
        Status = PciIo->Mem.Write (
            PciIo,                               // This
            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,                  // This
    EfiPciIoWidthUint32,    // Width
    1,                      // BarIndex
    0x10,                  // Offset
    BIT0,                  // Mask
    BIT0,                  // Value
    EFI_TIMER_PERIOD_SECONDS (1), // Timeout
    &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 += NumberOfBytes;
*Length     -= NumberOfBytes;
} while (*Length != 0);
return Status;
}

```

Example 178—Bus master read operation

18.5.7 DMA Bus Master Write Operation

The general algorithm for performing a bus master write operation 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. If not, add an extra read.

- Call `Flush()`.
- Call `Unmap()`.
- The processor may read the contents of the DMA buffer using `HostAddress`.

The following example 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 written by the bus master is specified by `HostAddress` and `Length`. This function performs 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 the function uses to check whether 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 used in this example include `Map()`, `Unmap()`, `Mem.Write()`, `PollMem()`, and `Flush()`.

A 32-bit PCI bus master is used for this example. A 64-bit PCI bus master would involve 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, the `EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE` attribute must be set in the Driver Binding Protocol `Start()` service of the PCI driver.

```
#include <Uefi.h>
#include <Protocol/PciIo.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI
DoBusMasterWrite (
    IN EFI_PCI_IO_PROTOCOL    *PciIo,
    IN UINT8                  *HostAddress,
    IN UINTN                  *Length
)
{
    EFI_STATUS          Status;
    UINTN              NumberOfBytes;
    EFI_PHYSICAL_ADDRESS DeviceAddress;
    VOID               *Mapping;
    VOID               DmaStartAddress;
    UINT64             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,                               // This
            EfiPciIoOperationBusMasterWrite,     // Operation
            (VOID *)HostAddress,                // HostAddress
            &NumberOfBytes,                     // NumberOfBytes
        );
    }
}
```

```

        &DeviceAddress,           // DeviceAddress
        &Mapping                 // 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,                  // This
    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 also starts the DMA transaction
//
Status = PciIo->Mem.Write (
    PciIo,                  // This
    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
// The MMIO read operations performed by PollMem() also flush all posted
// writes from the PCI bus master and through PCI-to-PCI bridges.
//
Status = PciIo->PollMem (
    PciIo,                  // This
    EfiPciIoWidthUint32,    // Width
    1,                      // BarIndex
    0x10,                  // Offset
    BIT0,                   // Mask
    BIT0,                   // Value
    EFI_TIMER_PERIOD_SECONDS (1), // Timeout
    &ControllerStatus       // Result
);
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 += NumberOfBytes;
*Length      -= NumberOfBytes;
} while (*Length != 0);
return Status;
}

```

Example 179—Bus master write operation

18.5.8 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()` 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 [example below](#) shows an example function the `start()` service calls to set up a common buffer operation for a specific PCI controller. The function accesses the PCI controller through the `PciIo` parameter. The function 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 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 Driver Binding Protocol `Start()` service of the PCI driver.

```

#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS
EFIAPI
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,                                // This
        AllocateAnyPages,                      // Type
        EfiBootServicesData,                  // MemoryType
        EFI_SIZE_TO_PAGES (Length),           // Pages
        (VOID **)HostAddress,                // HostAddress
        0                                     // Attributes
    );
    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,                                // This
        EfiPciIoOperationBusMasterCommonBuffer, // Operation
        (VOID *)HostAddress,                  // HostAddress
        &NumberOfBytes,                      // NumberOfBytes
        &DeviceAddress,                     // DeviceAddress
        Mapping                             // 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 offset 0x18 of Bar #1
    //
    DmaStartAddress = (UINT32)DeviceAddress;
    Status = PciIo->Mem.Write (
        PciIo,                                // This

```

```

        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 180—Allocate bus master common buffer

This example shows a function the `Stop()` service calls to free a common buffer for a PCI controller. The function accesses the PCI controller through the services of the `PciIo` parameter and uses them to free the common buffer specified by `HostAddress` and `Length`. This function undoes the mapping and frees the common buffer. The services of the PCI I/O Protocol used in this example include `Unmap()` and `FreeBuffer()`.

```

#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS
EFIAPI
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 181—Free bus master common buffer

18.6 PCI Optimization Techniques

Several techniques can be used to reduce size and optimize the performance of a UEFI Driver requiring access to PCI related resources. The following sections show examples of these techniques applicable to the services provided by the PCI I/O Protocol.

18.6.1 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 accessed through BAR #0 of the PCI video controller. The following four examples of performing this operation are shown from slowest to fastest:

The following two methods can significantly increase performance of a UEFI driver by taking advantage of the fill operations to eliminate loops and writing to a PCI controller at the largest possible size.

```
#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT8 Color8;
UINTN Index;

// This is the slowest method. It performs SIZE_1MB calls through PCI I/O and
// writes to the frame buffer 8 bits at a time.
//
Color8 = 0;
for (Index = 0; Index < SIZE_1MB; Index++) {
    Status = PciIo->Mem.Write (
        PciIo,
        EfiPciIoWidthUint8, // Width
        0, // BarIndex
        Index, // Offset
        1, // Count
        &Color8 // Buffer
    );
}
```

Example 182—PCI I/O 8-bit fill with a loop

```
#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT32 Color32;
UINTN Index;

// This is the slowest method. It performs SIZE_1MB calls through PCI I/O and
// writes to the frame buffer 8 bits at a time.
//
Color32 = 0;
for (Index = 0; Index < SIZE_1MB; Index += 4) {
    Status = PciIo->Mem.Write (
        PciIo,
        EfiPciIoWidthUint32, // Width
        0, // BarIndex
        Index, // Offset
        1, // Count
        &Color32 // Buffer
    );
}
```

Example 183—PCI I/O 32-bit fill with a loop

```

#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT8 Color8;

//
// 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;
Status = PciIo->Mem.Write (
    PciIo,                               // This
    EfiPciIoWidthFillUint8,             // Width
    0,                                  // BarIndex
    0,                                  // Offset
    SIZE_1MB,                           // Count
    &Color8                            // Buffer
);

```

Example 184—PCI I/O 8-bit fill without a loop

```

#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT32 Color32;

//
// 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;
Status = PciIo->Mem.Write (
    PciIo,                               // This
    EfiPciIoWidthFillUint32,            // Width
    0,                                  // BarIndex
    0,                                  // Offset
    SIZE_1MB / sizeof (UINT32),         // Count
    &Color32                            // Buffer
);

```

Example 185—PCI I/O 32-bit fill without a loop

18.6.2 PCI I/O FIFO operations

The examples below show 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 calls the PCI I/O Protocol once to perform the same operation, providing better performance if compiled with an EBC compiler. This example applies equally to FIFO read operations.

```

#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINTN Index;
UINT16 Buffer[256];

```

```

// 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,                                // This
        EfiPciIoWidthUint16,                  // Width
        EFI_PCI_IO_PASS_THROUGH_BAR,          // BarIndex
        0x1F0,                                // Offset
        1,                                     // Count
        &Buffer[Index]                      // Buffer
    );
}

```

Example 186—PCI I/O FIFO using a loop

```

#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS           Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT16              Buffer[256];

//
// This is the fastest method. It uses a loop to write 256 16-bit values to
// the IDE controller.
//
Status = PciIo->Io.Write (
    PciIo,                                // This
    EfiPciIoWidthFifoUint16,               // Width
    EFI_PCI_IO_PASS_THROUGH_BAR,           // BarIndex
    0x1F0,                                // Offset
    256,                                   // Count
    Buffer                                // Buffer
);

```

Example 187—PCI I/O FIFO without a loop

18.6.3 PCI I/O CopyMem() Operations

The following [examples](#) show how scrolling a frame buffer by different methods can provide performance improvements. In the first, the scroll operation is performed using a loop to move one scan line at a time. The PCI I/O Protocol `CopyMem()` service is similar to the UEFI Boot Service `CopyMem()`, except the PCI I/O Protocol operates on PCI MMIO ranges described by PCI MMIO BARs.

In general, the PCI I/O Protocol should be used, whenever possible, to eliminate loops in the UEFI Driver. This example assumes a 1 MB frame buffer MMIO, accessed through BAR #0 of the PCI graphics controller, with a screen 800 pixels wide, and 32 bits per pixel.

In the second example, the scroll operation is performed using a single PCI I/O Protocol call to `CopyMem()` to produce the exact same result. The second example executes significantly faster if the UEFI Driver is compiled with an EBC compiler because the loop has been removed from the UEFI Driver.

```

#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS           Status;

```

```

EFI_PCI_IO_PROTOCOL *PciIo;
UINTN ScanLineWidth;
UINTN Index;
UINT32 Value;

//
// This is the slowest method that moves one pixel at a time
// through the PCI I/O protocol.
//
ScanLineWidth = 800 * sizeof (UINT32);
for (Index = ScanLineWidth; Index < SIZE_1MB; Index += 4) {
    Status = PciIo->Mem.Read (
        PciIo,                                // This
        EfiPciIoWidthUint32,                  // Width
        0,                                     // BarIndex
        Index,                                // Offset
        1,                                     // Count
        &Value                                // Buffer
    );
    Status = PciIo->Mem.Write (
        PciIo,                                // This
        EfiPciIoWidthUint32,                  // Width
        0,                                     // Bar Index
        Index - ScanLineWidth,                // Offset
        1,                                     // Count
        &Value                                // Buffer
    );
}

```

Example 188—Scroll frame buffer using a loop

```

#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINTN ScanLineWidth;

//
// This is the faster method that makes a single call to CopyMem().
//
ScanLineWidth = 800 * sizeof (UINT32);
Status = PciIo->CopyMem (
    PciIo,                                // This
    EfiPciIoWidthUint32,                  // Width
    0,                                     // DestBarIndex
    0,                                     // DestOffset
    0,                                     // SrcBarIndex
    ScanLineWidth,                         // SrcOffset
    (SIZE_1MB / sizeof (UINT32)) - ScanLineWidth // Count
);

```

Example 189—Scroll frame buffer without a loop

18.6.4 PCI Configuration Header Operations

The following [three examples](#) demonstrate different methods to read a PCI configuration header from a PCI controller, ordered lowest to highest in performance. The first example uses a loop to read the header 8 bits at a time; the second uses a single call to read the entire header 8 bits at a time and the third uses a single call to read the header 32 bits at a time.

```
#include <Uefi.h>
#include <Protocol/PciIo.h>
#include <IndustryStandard/Pci.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
PCI_TYPE00 Pci;
UINT32 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,                      // This
        EfiPciIoWidthUint8,          // Width
        Index,                      // Offset
        1,                          // Count
        (UINT8 *)(&Pci) + Index    // Buffer
    );
}
```

Example 190—Read PCI configuration using a loop

```
#include <Uefi.h>
#include <Protocol/PciIo.h>
#include <IndustryStandard/Pci.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
PCI_TYPE00 Pci;

//
// This is a faster method that removes the loop and reads 8 bits at a time.
//
Status = PciIo->Pci.Read (
    PciIo,                      // This
    EfiPciIoWidthUint8,          // Width
    0,                          // Offset
    sizeof (Pci),              // Count
    &Pci                         // Buffer
);
```

Example 191—Read PCI configuration 32 bits at a time

```
#include <Uefi.h>
#include <Protocol/PciIo.h>
#include <IndustryStandard/Pci.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
PCI_TYPE00 Pci;

//
// 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,                      // This
    EfiPciIoWidthUint32,          // Width
    0,                          // Offset
    sizeof (Pci) / sizeof (UINT32), // Count
    &Pci                         // Buffer
);
```

Example 192—Read PCI configuration 32 bits at a time

18.6.5 PCI I/O MMIO Buffer 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. 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 UEFI Driver for a graphics controller.

Note: *The examples shown here apply equally well to reading a bitmap from the frame buffer of a PCI video controller using the `PciIo->Mem.Read()` function.*

```
#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT8 gBitMap[SIZE_1MB];
UINTN Index;

//
// Loop writing a 1 MB bitmap to the frame buffer 8 bits at a time.
//
for (Index = 0; Index < sizeof (gBitMap); Index++) {
    Status = PciIo->Mem.Write (
        PciIo,
        EfiPciIoWidthUint8, // This
        0,                  // Width
        0,                  // BarIndex
        Index,              // Offset
        1,                  // Count
        &gBitMap[Index]     // Buffer
    );
}
```

Example 193—Write 1MB Frame Buffer using a loop

```
#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT8 gBitMap[SIZE_1MB];

//
// Faster method that removes the loop and writes 32 bits at a time.
//
Status = PciIo->Mem.Write (
    PciIo,
    EfiPciIoWidthUint32, // This
    0,                  // Width
    0,                  // BarIndex
    0,                  // Offset
    sizeof (gBitMap) / sizeof (UINT32), // Count
    gBitMap             // Buffer
);
```

Example 194—Write 1MB Frame Buffer with no loop

18.6.6 PCI I/O Polling Operations

These same types of optimization can be applied to polling as well. In the [following examples](#), two different polling methods are shown:

- A loop with 10 µs stalls to wait up to 1 minute

- A single call to PCI I/O protocol to perform the entire operation.

These types of polling operations are usually performed when a driver is waiting for the hardware to complete an operation with the completion status indicated by a bit changing state in an I/O port or a memory-mapped I/O port. The examples below poll offset 0x20 in BAR #1 for bit 0 to change from 0 to 1.

The `PollIo()` and `PollMem()` functions in the PCI I/O Protocol are very flexible and can simplify the operation of polling for bits to change state in status registers.

```
#include <Uefi.h>
#include <Protocol/PciIo.h>

EFI_STATUS Status;
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 be set.
//
for (TimeOut = 0; TimeOut < 1000000; TimeOut += 10) {
    Status = PciIo->Mem.Read (
        PciIo,                      // This
        EfiPciIoWidthUint8,          // Width
        1,                           // BarIndex
        0x20,                         // Offset
        1,                           // Count
        &Result8                     // Value
    );
    if ((Result8 & BIT0) == BIT0) {
        return EFI_SUCCESS;
    }
    gBS->Stall (10);
}

return EFI_TIMEOUT;
```

Example 195—Using Mem.Read() and Stall() to poll for 1 second

```
#include <Uefi.h>
#include <Protocol/PciIo.h>
#include <Library/UefiLib.h>

EFI_STATUS Status;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT64 Result64;

// 
// Call PollIo() to poll for Bit #0 in register 0x20 of Bar #1 to be set.
//
Status = PciIo->PollIo (
    PciIo,                      // This
    EfiPciIoWidthUint8,          // Width
    1,                           // BarIndex
    0x20,                         // Offset
    BIT0,                         // Mask
    BIT0,                         // Value
    EFI_TIMER_PERIOD_SECONDS (1), // Timeout
    &Result64                     // Result
);
```

Example 196—Using PollIo() to poll for 1 second

18.7 PCI Option ROM Images

The EDK II provides tools to aide in the development of UEFI drivers for PCI adapters. Once UEFI Driver(s) for a PCI adapter are built, they need to be packaged into PCI option ROM compatible image format. UEFI drivers stored in PCI option ROMs are automatically loaded and executed by the PCI bus driver during PCI enumeration.

The EDK II tools provide two methods to generate a PCI Option ROM image. These are the **EfiRom** utility and the EDK II INF/FDF file syntax.

Using the **build** command, each allows a UEFI Driver developer to describe how UEFI Drivers should be packaged into a PCI Option ROM image as part of the standard EDK II build process.

Use either PCI Option ROM image with a PROM programmer or a flash update utility to reprogram the PCI option ROM container on a PCI adapter.

18.7.1 EfiRom Utility

The **EfiRom** utility is included with the standard set of tools from the EDK II project. A pre-built binary of **EfiRom** is in the **BaseTools/Bin/Win32** directory in the EDK II **WORKSPACE**. This directory, with pre-built binaries, is automatically added to the path after setting up the EDK II environment, so **EfiRom** is always available.

The sources to **EfiRom** are in the **BaseTools/Source/C/EfiRom** directory so the utility can be built for any operating system supporting the EDK II.

Use the **EfiRom** utility to build PCI Option ROM Images containing UEFI Drivers, PC BIOS legacy option ROM images, or both, in a format conforming to the *PCI 2.3 Specification* and *PCI 3.0 Specification*. The **EfiRom** utility also allows UEFI Drivers to be compressed using the UEFI compression algorithm defined in the Compression Algorithm Specification section of the *UEFI Specification*.

The **EfiRom** utility performs some rudimentary checks on the UEFI Drivers to verify they are valid PE/COFF images as defined by the *Microsoft Portable Executable and Common Object File Format Specification*. If any of these checks fail, the utility aborts without creating the output ROM image file. For example, the following checks are performed on UEFI Drivers:

- Verification that the DOS stub magic value is **0x5A4D**
- Verification that the PE signature is “**PE\0\0**”
- The **EfiRom** utility also performs rudimentary checking of PC BIOS legacy option ROM images. If any of these checks fail, the utility aborts without creating the output ROM image file. The following checks are performed on PC BIOS legacy option ROMs:
 - Verification that the signature of the option ROM header is **0xAA55**
 - Verification that the offset to the PCI data structure is within the range of the file size.
 - Verification that the signature of the PCI data structure is “**PCIR**”.

This example shows the help information from the **EfiRom** utility displayed when the utility is run with no input parameters, the **-h** option or the **--help** option.

```
Usage: EfiRom -f VendorId -i DeviceId [options] [file name<s>]

Copyright (c) 2007 - 2011, Intel Corporation. All rights reserved.

Options:
  -o FileName, --output FileName
    File will be created to store the output content.
  -e EfiFileName
    EFI PE32 image files.
  -ec EfiFileName
    EFI PE32 image files and will be compressed.
  -b BinFileName
    Legacy binary files.
  -l ClassCode
    Hex ClassCode in the PCI data structure header.
  -r Rev    Hex Revision in the PCI data structure header.
  -n        Not to automatically set the LAST bit in the last file.
  -f VendorId
    Hex PCI Vendor ID for the device OpROM, must be specified
  -i DeviceId
    Hex PCI Device ID for the device OpROM, must be specified
  -p, --pci23
    Default layout meets PCI 3.0 specifications
    specifying this flag will for a PCI 2.3 layout.
  -d, --dump
    Dump the headers of an existing option ROM image.
  -v, --verbose
    Turn on verbose output with informational messages.
  --version Show program's version number and exit.
  -h, --help
    Show this help message and exit.
  -q, --quiet
    Disable all messages except FATAL ERRORS.
  --debug [#,0-9]
    Enable debug messages at level #.
```

Example 197—EfiRom Utility Help

Examples of generating an Option ROM image using various options provided by the **EfiRom** utility follow:

Generate a PCI Option ROM image with a single UEFI binary files.

The output filename is not specified in command line, so the output filename is **File2.rom**. The output filename is the same as the first input filename with the extension **.rom**. When UEFI binary files are specified, the VendorId flag **-f** and DeviceId flag **-i** must be specified.

```
EfiRom -f 0xABCD -i 0x1234 -e File2.efi
```

This example shows the output of the **EfiRom** utility then the **-d** option is used to display the headers from the PCI Option ROM image generated in the previous example.

Image 1 -- Offset 0x0	
ROM header contents	
Signature	0xAA55

PCIR offset	0x001C
Signature	PCIR
Vendor ID	0xABCD
Device ID	0x1234
Length	0x001C
Revision	0x0003
DeviceListOffset	0x00
Class Code	0x000000
Image size	0x1800
Code revision:	0x0000
MaxRuntimeImageLength	0x00
ConfigUtilityCodeHeaderOffset	0x00
DMTFCLPEntryPointOffset	0x00
Indicator	0x80 (last image)
Code type	0x03 (EFI image)
EFI ROM header contents	
EFI Signature	0x0EFI
Compression Type	0x0000 (not compressed)
Machine type	0x014C (IA32)
Subsystem	0x000B (EFI boot service driver)
EFI image offset	0x0038 (@0x38)

Example 198—EfiRom Utility Dump Feature

Generate a PCI Option ROM image with two UEFI binary files and one PC BIOS legacy option ROM binary file.

The output filename is not specified in command line so the output filename is **File1.rom**. The output filename is the same as the first input filename with the extension **.rom**. When UEFI binary files are specified, the VendorId flag **-f** and DeviceId flag **-i** must be specified.

```
EfiRom -f 0xABCD -i 0x1234 -e File1.efi File2.efi -b Legacy.bin
```

Generate a PCI Option ROM image with two UEFI binary files and one PC BIOS legacy option ROM binary file with the output filename specified on the command line as **File.rom.**

When UEFI binary files are specified, the VendorId flag **-f** and DeviceId flag **-i** must be specified.

```
EfiRom -o File.rom -f 0xABCD -i 0x1234 -e File1.efi File2.efi -b Legacy.bin
```

Generate a PCI Option ROM image with two UEFI binary files and one PC BIOS legacy option ROM binary file.

The output filename is specified in command line as **Compressed.rom**. UEFI binary files are compressed using the UEFI Compression algorithm. When UEFI binary files are specified, the VendorId flag **-f** and DeviceId flag **-i** must be specified.

```
EfiRom -o Compressed.rom -f 0xABCD -i 0x1234 -ec File1.efi File2.efi -b Legacy.bin
```

18.7.2 Using INF File to Generate PCI Option ROM Image

Use the INF file to specify the information required to package a UEFI Driver into a PCI Option ROM image without having to manually run the **EfiRom** utility. [Chapter 7](#) covers Driver Entry Points and includes a number of example INF files. The [following example](#)

shows an expanded version of the `AbcDriverMinimum` from [Chapter 7](#) and also shows how the PCI Option ROM related information can be specified. The only changes are the addition of the PCI statements in the [Defines] section. These PCI statements allow the Vendor ID, Device ID, Class Code, and Revision values to be specified and they are used to fill in the PCI Option ROM headers. The `PCI_COMPRESS` statement specifies whether the UEFI Driver should be compressed using the UEFI compression algorithm or not. If a statement is not present, the value is assumed to be 0. If the PCI statements are present, and if the UEFI Driver is successfully built, the PCI Option ROM image is then automatically generated. The one limitation of this method is that the PCI Option ROMs are allowed to contain only a single UEFI Driver.

```
[Defines]
INF_VERSION      = 0x00010005
BASE_NAME        = AbcDriverPciOptionRom
FILE_GUID        = DA87D340-15C0-4824-9BF3-D52286674BEF
MODULE_TYPE       = CAE55A8A-4307-4ae1-824E-326EE24928D7
VERSION_STRING   = 1.0
ENTRY_POINT      = AbcDriverEntryPoint
PCI_VENDOR_ID    = 0xABCD
PCI_DEVICE_ID    = 0x1234
PCI_CLASS_CODE   = 0x56789A
PCI_REVISION     = 0x0003
PCI_COMPRESS     = TRUE

[Sources]
Abc.c

[Packages]
MdePkg/MdePkg.dec

[LibraryClasses]
UefiDriverEntryPoint
```

Example 199—UEFI Driver INF File for PCI Option ROM

18.7.3 Using FDF File to Generate PCI Option ROM Image

When managing large numbers of UEFI Drivers and PCI Option ROMs, greater flexibility than the `EfiRom` utility or the INF methods allow may be required. The EDK II build system supports an FDF file format that provides methods to package UEFI Drivers into FLASH devices. The FDF file format also supports the description of PCI Option ROMs. The EDK II build system requires a DSC file to build UEFI Drivers. The DSC file format is covered in more detail in [Chapter 30](#). A DSC file can optionally specify an associated FDF file in the [Defines] section of the DSC file with a `FLASH_DEFINITION` statement. The example below shows the [Defines] section of a DSC file specifying the FDF file `AbcDriver.fdf`. The FDF file is typically in the same directory as the DSC file.

```
[Defines]
PLATFORM_NAME          = AbcDriver
PLATFORM_GUID           = 14893C02-5693-47ab-AEF5-61DFA089508A
PLATFORM_VERSION         = 0.10
DSC_SPECIFICATION       = 0x00010005
OUTPUT_DIRECTORY         = Build/AbcDriver
SUPPORTED_ARCHITECTURES = IA32|IPF|X64|EBC|ARM
```

BUILD_TARGETS	= DEBUG RELEASE
SKUID_IDENTIFIER	= DEFAULT
FLASH_DEFINITION	= AbcDriver/AbcDriver.fdf

Example 200—Specify name of FDF file from a DSC file

The FDF file may describe one or more PCI Option ROM images. Unlike the INF method, Option ROM images are not limited to a single UEFI Driver. The following example shows an FDF file that produces three PCI Option ROM images called [AbcDriverAll.rom](#), [AbcDriverIA32.rom](#), and [AbcDriverX64.rom](#). The first PCI Option ROM image contains a UEFI Driver image compiled for IA32 and a UEFI Driver image compiled for X64. The syntax for specifying the PCI related definitions is the same as the INF example in the previous section. The second PCI Option ROM image contains only one UEFI Driver compiled for IA32. The third image contains one UEFI Driver compiled for X64. The UEFI Drivers are compressed in all three of these option ROM images.

```
[Rule.Common.UEFI_DRIVER]
FILE DRIVER = $(NAMED_GUID) {
    PE32 PE32 | .efi
}

[OptionRom.AbcDriverAll]
INF USE=IA32 AbcDriver/Abc.inf {
    PCI_VENDOR_ID = 0xABCD
    PCI_DEVICE_ID = 0x1234
    PCI_CLASS_CODE = 0x56789A
    PCI_REVISION = 0x0003
    PCI_COMPRESS = TRUE
}
INF USE=X64 AbcDriver/Abc.inf {
    PCI_VENDOR_ID = 0xABCD
    PCI_DEVICE_ID = 0x1234
    PCI_CLASS_CODE = 0x56789A
    PCI_REVISION = 0x0003
    PCI_COMPRESS = TRUE
}

[OptionRom.AbcDriverIAa32]
INF USE=IA32 AbcDriver/Abc.inf {
    PCI_VENDOR_ID = 0xABCD
    PCI_DEVICE_ID = 0x1234
    PCI_CLASS_CODE = 0x56789A
    PCI_REVISION = 0x0003
    PCI_COMPRESS = TRUE
}

[OptionRom.AbcDriverX64]
INF USE=X64 AbcDriver/Abc.inf {
    PCI_VENDOR_ID = 0xABCD
    PCI_DEVICE_ID = 0x1234
    PCI_CLASS_CODE = 0x56789A
    PCI_REVISION = 0x0003
    PCI_COMPRESS = TRUE
}
```

Example 201—Using an FDF file to Generate PCI Option ROM images

USB Driver Design Guidelines

There are several categories of USB drivers that cooperate to provide the USB driver stack in a platform. The table below lists these USB drivers.

Table 30—Classes of USB drivers

Class of driver	Description
USB 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 required to boot an EFI-conformant operating system.

This chapter shows how to write host controller drivers and USB device drivers. USB drivers must follow all of the general design guidelines described in [Chapter 4](#) of this guide. In addition, any USB host controllers that are PCI controllers must also follow the PCI-specific design guidelines (see [Chapter 18](#)).

Note: *USB device drivers do not typically include HII functionality because they do not have configurable information. For example, USB device drivers are typically for hot-plug devices.*

The [following figure](#) shows an example of a USB driver stack and the protocols 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 21.

In this example, the platform hardware provides a single USB host controller on the PCI bus. The PCI bus driver produces a handle with `EFI_DEVICE_PATH_PROTOCOL` and `EFI_PCI_IO_PROTOCOL` installed for this USB host controller. The USB host controller driver then consumes `EFI_PCI_IO_PROTOCOL` on that USB host controller device handle and installs 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 detects a USB keyboard, a USB mouse, and a USB mass storage device. As a result, the USB bus driver creates three child handles and installs the `EFI_DEVICE_PATH_PROTOCOL` and `EFI_USB_IO_PROTOCOL` onto each of those handles.

The USB mouse driver consumes the `EFI_USB_IO_PROTOCOL` and produces the `EFI_SIMPLE_POINTER_PROTOCOL`. The USB keyboard driver consumes the `EFI_USB_IO_PROTOCOL` to produce the `EFI_SIMPLE_TEXT_INPUT_PROTOCOL`. The USB mass

storage driver consumes the `EFI_USB_IO_PROTOCOL` to produce the `EFI_BLOCK_IO_PROTOCOL`.

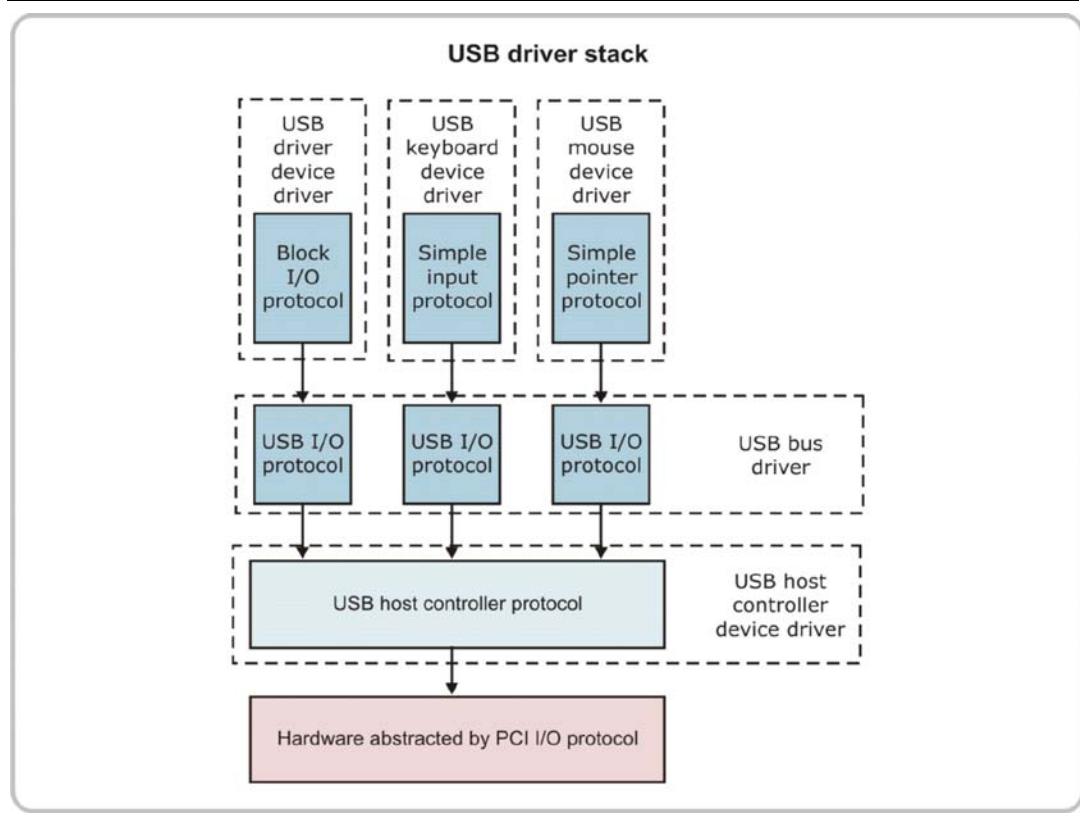


Figure 21—USB driver stack

The protocol interfaces for the USB2 Host Controller Protocol and the USB I/O Protocol are shown below in Example 202, below, and Example [203](#) following that.

```
typedef struct _EFI_USB2_HC_PROTOCOL EFI_USB2_HC_PROTOCOL;
{
    /**
     * The EFI_USB2_HC_PROTOCOL provides USB host controller management, basic
     * data transactions over a USB bus, and USB root hub access. A device driver
     * that wishes to manage a USB bus in a system retrieves the EFI_USB2_HC_PROTOCOL
     * instance that is associated with the USB bus to be managed. A device handle
     * for a USB host controller minimally contains an EFI_DEVICE_PATH_PROTOCOL
     * instance, and an EFI_USB2_HC_PROTOCOL instance.
     */
    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_ISOCRONOUS_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;

///
/// The major revision number of the USB host controller. The revision
/// information indicates the release of the Universal Serial Bus Specification
/// with which the host controller is compliant.
///
UINT16 MajorRevision;

///
/// The minor revision number of the USB host controller. The revision
/// information indicates the release of the Universal Serial Bus Specification
/// with which the host controller is compliant.
///
UINT16 MinorRevision;
};

```

Example 202—USB 2 Host Controller Protocol

```

typedef struct _EFI_USB_IO_PROTOCOL    EFI_USB_IO_PROTOCOL;

///
/// The EFI_USB_IO_PROTOCOL provides four basic transfers types described
/// in the USB 1.1 Specification. These include control transfer, interrupt
/// transfer, bulk transfer and isochronous transfer. The EFI_USB_IO_PROTOCOL
/// also provides some basic USB device/controller management and configuration
/// interfaces. A USB device driver uses the services of this protocol to manage
/// USB devices.
///
struct _EFI_USB_IO_PROTOCOL {
    //
    // IO transfer
    //
    EFI_USB_IO_CONTROL_TRANSFER           UsbControlTransfer;
    EFI_USB_IO_BULK_TRANSFER              UsbBulkTransfer;
    EFI_USB_IO_ASYNC_INTERRUPT_TRANSFER   UsbAsyncInterruptTransfer;
    EFI_USB_IO_SYNC_INTERRUPT_TRANSFER   UsbSyncInterruptTransfer;
    EFI_USB_IO_ISOCHRONOUS_TRANSFER     UsbIsochronousTransfer;
    EFI_USB_IO_ASYNC_ISOCHRONOUS_TRANSFER UsbAsyncIsochronousTransfer;

    //
    // Common device request
    //
    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_LANGUAGE   UsbGetSupportedLanguages;

    //
    // Reset controller's parent port
    //
    EFI_USB_IO_PORT_RESET               UsbPortReset;
};

```

Example 203—USB I/O Protocol

19.1 USB Host Controller Driver

The USB host controller driver depends on which USB host controller specification the host controller is based. Currently, the major types of USB host controllers are the following:

- Open Host Controller Interface (OHCI) (USB 1.0 and USB 1.1)
- Universal Host Controller Interface (UHCI) (USB 1.0 and USB 1.1)
- Enhanced Host Controller Interface (EHCI) (USB 2.0)
- Extended Host Controller Interface (XHCI) (USB 3.0)

The USB host controller driver is a device driver and 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. The EDK II provides UEFI Drivers that implement the `EFI_USB_HC2_PROTOCOL` for UHCI, EHCI, and XHCI in the MdeModulePkg in the following paths:

- UHCI - `MdeModulePkg/Bus/Pci/UhciDxe`
- EHCI - `MdeModulePkg/Bus/Pci/EhciDxe`
- XHCI - `MdeModulePkg/Bus/Pci/XhciDxe`

19.1.1 Driver Binding Protocol Supported()

The USB host controller driver must implement the `EFI_DRIVER_BINDING_PROTOCOL` containing the `Supported()`, `Start()`, and `Stop()` services. The Driver Binding Protocol is installed into the Handle Database in the drive entry point.

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, the `Supported()` service returns `EFI_SUCCESS`. Otherwise, the `Supported()` service returns `EFI_UNSUPPORTED`.

The [following example](#) shows an example of the `Supported()` service for the USB host controller driver managing a PCI controller with Class code 0x30c.

First, it attempts to open the PCI I/O Protocol `EFI_OPEN_PROTOCOL_BY_DRIVER`. If the PCI I/O Protocol cannot be opened, then the USB host controller driver does not support the controller specified by `ControllerHandle`. If the PCI I/O Protocol is opened, 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 fields match.

```

#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/PciIo.h>
#include <IndustryStandard/Pci.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS
EFIAPI
AbcSupported (
    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;
    UINT8              PciClass[3];

    //
    // 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 3 bytes of class code information from the PCI configuration header
    // at offset 0x09
    //
    Status = PciIo->Pci.Read (
        PciIo,
        EfiPciIoWidthUint8,           // This
        PCI_CLASSCODE_OFFSET,         // Width
        sizeof (PciClass),           // Offset
        &PciClass,                  // Count
        &PciClass                    // Buffer
    );
    if (EFI_ERROR (Status)) {
        goto Done;
    }

    //
    // Test whether the class code is for a USB UHCI controller
    //
    if ((PciClass[2] != PCI_CLASS_SERIAL) ||
        (PciClass[1] != PCI_CLASS_SERIAL_USB) ||
        (PciClass[0] != PCI_IF_UHCI)) {
        Status = EFI_UNSUPPORTED;
    }
}

Done:
//
// Close the PCI I/O Protocol
//
gBS->CloseProtocol (
    ControllerHandle,
    &gEfiPciIoProtocolGuid,
    This->DriverBindingHandle,
    ControllerHandle
)

```

```

    );
    return Status;
}

```

Example 204—Supported() service for USB host controller driver

19.1.2 Driver Binding Protocol Start()

The `start()` service of the Driver Binding Protocol for the USB host controller driver also opens the PCI I/O Protocol with an attribute of `EFI_OPEN_PROTOCOL_BY_DRIVER`. This is followed by an initialization of the USB host controller hardware and an installation of a `EFI_USB2_HC_PROTOCOL` instance into the Handle Database.

19.1.2.1 Support for legacy devices

Some USB host controllers provide legacy support to be compatible with legacy devices. Under this mode, the USB input device, including mouse and keyboard, act as if they are behind an 8042 keyboard controller. A UEFI implementation uses the native USB support rather than the legacy support.

As a result, the USB legacy support must be disabled in the `start()` service of the USB host controller driver, before enabling the USB host controller. This step is required because the legacy support conflicts with the native USB support provided in UEFI USB driver stack. The example below shows how to turn off USB legacy support for a UHCI 1.1 Host Controllers.

```

///
/// USB legacy Support
///
#define USB_EMULATION 0xc0

EFI_STATUS
EFIAPI
TurnOffUSBLegacySupport (
    IN EFI_PCI_IO_PROTOCOL *PciIo
)
{
    EFI_STATUS Status;
    UINT16 Command;

    //
    // Disable USB Legacy Support by writing 0x0000 to the USB_EMULATION
    // register in the PCI Configuration of the PCI Controller
    //
    Command = 0;
    Status = PciIo->Pci.Write (
        PciIo,
        EfiPciIoWidthUint16, // Width
        USB_EMULATION, // Offset
        1, // Count
        &Command // Buffer
    );
    return Status;
}

```

Example 205—Disable USB Legacy Support

19.1.3 Driver Binding Protocol Stop()

The `stop()` service must perform the reverse of the steps the `start()` service performs. The USB host controller driver is required to make sure that there are no memory leaks or handle leaks, as well as making sure that hardware is stopped accordingly, including restoration of the PCI I/O Protocol attributes as described in [Chapter 18](#).

19.1.4 USB 2 Host Controller Protocol Data Transfer 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 transactions over the Universal Serial Bus. It also provides services to manage the USB root hub integrated into the USB host controller.

[Appendix A](#) provides a template for the implementation of the USB Host Controller Protocol. The services of the USB 2 Host Controller Protocol can be categorized into the following categories:

- 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 root hub-related services and host controller state-related services, implementation mainly involves read/write operations to specific USB host controller registers. The USB host controller data sheet provides information on these register usages, so this topic is not covered in detail here.

This section concentrates on the USB transfer-related services. Those transfers are categorized as either *asynchronous* or *synchronous*.

With asynchronous transfers, the transfer does not complete with the service's return. With synchronous transfers, the requested transfer has completed when the service returns. The following sections discuss these two types of transfers in more detail.

19.1.4.1 Synchronous transfer

The USB Host Controller Protocol provides the following four synchronous transfer services:

- `ControlTransfer()`
- `BulkTransfer()`
- `SyncInterruptTransfer()`
- `IsochronousTransfer()`

Control and bulk transfers are completed in an acceptable period of time and thus are natural synchronous transfers in the view of an UEFI 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 synchronous transfer services because there is no worry about when the data is ready. The transfer result is 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, implementing a bulk transfer service 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 decides the transfer direction and sets up the PCI bus master read operation or write operation. For example, if the transfer direction is `EfiUsbDataIn`, the USB host controller reads from the DMA buffer. A bus master write operation is required.
- **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 used to support the requirements of control, bulk, and interrupt transfers.

In this step, these QH and TD structures are created and linked to the Frame List. One possible implementation can be creation of one QH and a list of TDs to form a transfer list. The QH points to the first TD and occupies one entry in the Frame List.

- **Executing the TD and getting the result:** The USB host controller automatically executes the TD when the timer expires. The UHCI driver waits until all of the TDs associated with the transfer are all completed. After that, the result of the TD execution is determined.
- **Cleaning up:** Delete the bulk transfer QH and TD structures from the Frame List, free related structures, and unmap the PCI DMA operation.

19.1.4.2 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 registers a periodic timer event. Meanwhile, it maintains a queue for all asynchronous transfers. When the timer event is signaled, the timer event callback function evaluates this queue and checks to see if asynchronous transfers are now complete.

Generally speaking, the main work of the timer event callback function is to go through the asynchronous transfers queue. For each asynchronous transfer, it checks whether an asynchronous transfer is completed or not and performs the following:

- **If not completed:** The USB host controller driver takes no action and leaves the transfer on the queue.
- **If completed:** The USB host controller driver copies the data that it received to a predefined data buffer and removes the related QH and TD structures. It also invokes a preregistered transfer callback function. Based on that transfer's complete status, the USB host controller driver takes different additional actions such as:
 - If completed without error, update the transfer data status accordingly, e.g., data toggle bit.
 - If completed with error, it is suggested that the USB host controller do nothing and leave the error recovery work to the related USB device driver.

19.1.4.3 Internal Memory Management

To implement USB transfers, the USB host controller driver manages many small memory fragments as transfer data (i.e. QH and TD). If the USB host controller driver uses the system memory management services to allocate these memory fragments each time, then the overhead can be large. As a result, it is recommended that the USB host controller driver manage these kinds of internal memory usage itself. One possible implementation, as in EDK II, is that the host controller driver can allocate a large buffer of memory in the Driver Binding Protocol `start()` service using UEFI memory services. The USB host controller driver provides 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.

19.1.4.4 DMA

Most USB host controllers use DMA for their data transfer between host and devices. Because the processor and USB host controller both access that transfer data simultaneously, the USB host controller driver must 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 18](#) for usage guidelines for managing PCI DMA for common buffers.

19.2 USB Bus Driver

EDK II 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](#). The implementation of the USB Bus Driver is found in the [MdeModulePkg](#) in the directory [MdeModulePkg/Bus/Usb/UsbBusDxe](#).

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 USB device drivers are required for USB hub devices.

If UEFI-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. If new types of USB devices are introduced that provide console or UEFI boot capabilities, the implementation of new USB Device Drivers is also required.

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 detailed design and implementation of the USB bus driver is not covered in this guide

19.3 USB Device Driver

USB device drivers 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 must follow the UEFI Driver Model. As mentioned above, the USB device drivers do not manage hub devices because those hub devices are managed by the USB bus driver. The EDK II provides a number of USB Device Drivers in the [MdeModulePkg](#) for devices that are typically used to provide UEFI consoles and UEFI boot devices. The EDK II [MdePkg](#) also provides a library called [UefiUsbLib](#) that provides functions to simplify the implementations of USB device drivers using the USB I/O Protocol. Some of the USB Device Driver implementations provided in the EDK II are as follows:

- USB Keyboard: [MdeModulePkg/Bus/Usb/UsbKbDxe](#)
- USB Mouse: [MdeModulePkg/Bus/Usb/UsbMouseDxe](#)
- USB Mouse: [MdeModulePkg/Bus/Usb/UsbMouseAbsolutePointerDxe](#)
- USB Mass Storage: [MdeModulePkg/Bus/Usb/UsbMassStorageDxe](#)

19.3.1 Driver Binding Protocol Supported()

USB device drivers must implement the `EFI_DRIVER_BINDING_PROTOCOL` that contains the `Supported()`, `Start()`, and `Stop()` services. The `Supported()` service checks the passed in controller handle to determine whether this handle represents a USB device that the 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 the USB device. Check whether the values of this device's `InterfaceClass`, `InterfaceSubClass`, and `InterfaceProtocol` are identical to the corresponding values this driver can manage.

If the above two checks are passed, the USB device driver can manage the device the controller handle represents and the `Supported()` service returns `EFI_SUCCESS`.

Otherwise, the `Supported()` service returns `EFI_UNSUPPORTED`. In addition, this check process must not disturb the current state of the USB device because the USB device may be managed by another USB device driver.

The example below shows an implementation of the Driver Binding Protocol `Supported()` service for a USB keyboard driver. It opens the USB I/O Protocol with an attribute of `EFI_OPEN_PROTOCOL_BY_DRIVER`. It then uses the `UsbGetInterfaceDescriptor()` service of the USB I/O Protocol and evaluates the class, subclass, and protocol fields of the interface descriptor to see if the description is for a USB keyboard.

```
#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/UsbIo.h>
#include <Library/UefiBootServicesTableLib.h>

#define CLASS_HID          3
#define SUBCLASS_BOOT       1
#define PROTOCOL_KEYBOARD   1

EFI_STATUS
EFIAPI
AbcSupported (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE                  ControllerHandle,
    IN EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath OPTIONAL
)
{
    EFI_STATUS                      Status;
    EFI_USB_IO_PROTOCOL             *UsbIo;
    EFI_USB_INTERFACE_DESCRIPTOR    InterfaceDescriptor;

    //
    // Open the USB I/O Protocol on ControllerHandle
    //
    Status = gBS->OpenProtocol (
        ControllerHandle,
        &gEfiUsbIoProtocolGuid,
        (VOID **)&UsbIo,
        This->DriverBindingHandle,
        ControllerHandle,
        EFI_OPEN_PROTOCOL_BY_DRIVER
    );
    if (Status != EFI_SUCCESS) {
        return Status;
    }

    // Check if the device is a keyboard
    if (UsbIo->InterfaceDescriptor->InterfaceClass != CLASS_HID) {
        return EFI_UNSUPPORTED;
    }
    if (UsbIo->InterfaceDescriptor->InterfaceSubClass != SUBCLASS_BOOT) {
        return EFI_UNSUPPORTED;
    }
    if (UsbIo->InterfaceDescriptor->InterfaceProtocol != PROTOCOL_KEYBOARD) {
        return EFI_UNSUPPORTED;
    }

    return EFI_SUCCESS;
}
```

```

        );
if (EFI_ERROR (Status)) {
    return Status;
}

//
// Get the USB Interface Descriptor
//
Status = UsbIo->UsbGetInterfaceDescriptor (
    UsbIo,
    &InterfaceDescriptor
);
if (EFI_ERROR(Status)) {
    goto Done;
}

//
// Check to see if the interface descriptor is supported by this driver
//
if (InterfaceDescriptor.InterfaceClass != CLASS_HID
    || InterfaceDescriptor.InterfaceSubClass != SUBCLASS_BOOT
    || InterfaceDescriptor.InterfaceProtocol != PROTOCOL_KEYBOARD) {
    Status = EFI_UNSUPPORTED;
}

Done:
//
// Close the PCI I/O Protocol
//
gBS->CloseProtocol (
    ControllerHandle,
    &gEfiUsbIoProtocolGuid,
    This->DriverBindingHandle,
    ControllerHandle
);

return Status;
}

```

Example 206—Supported() for a USB device driver

Because the `Supported()` service is invoked many times, the USB bus driver in EDK II makes certain optimizations. The USB bus driver caches the interface descriptors, eliminating the need to read them from the USB device every time a USB device driver's `Supported()` service is invoked.

19.3.2 Driver Binding Protocol Start() and Stop()

The `start()` service of the Driver Binding Protocol for a USB device driver opens the USB I/O Protocol with an attribute of `EFI_OPEN_PROTOCOL_BY_DRIVER`. The service then installs the I/O abstraction protocol for the USB device or host controller onto the handle on which the `EFI_USB_IO_PROTOCOL` is installed.

19.3.2.1 Example using a USB mass storage device

This discussion provides detailed guidance on how to implement a USB device driver. It uses a USB mass storage device as an example. For 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 discussions cover how to implement the `Start()` and `Stop()` driver binding protocol services and UEFI Block I/O protocol.

This example shows a portion of the private context data structure for a USB mass storage device driver. See [Chapter 8](#) of this guide for more information about design guidelines for private context data structures.

```
#include <Uefi.h>
#include <Protocol/UsbIo.h>
#include <Protocol/BlockIo.h>

typedef struct {
    UINT64                               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 207—USB mass storage driver private context data structure

19.3.2.2 Example implementing Driver Binding Start()

The following steps are performed in the Driver Binding Protocol `Start()` service.

1. Open the USB I/O Protocol on `ControllerHandle` `EFI_OPEN_PROTOCOL_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 allocates memory for the private data structure and does the required initializations—for example, setting up the `Signature`, `UsbIo`, and `InterfaceDescriptor` fields.
4. **Parse the interface descriptor.** In this step, the USB device driver parses the `InterfaceDescriptor` that was obtained in step 2, and verifies that all bulk and interrupt endpoints exist. The `NumEndpoints` field in `InterfaceDescriptor` indicates how many endpoints are in this USB interface. Next, the endpoint descriptors are retrieved one by one by using the `UsbGetEndpointDescriptor()` service. Then, the `Attributes` and `EndpointAddress` fields in `EndpointDescriptor` are evaluated to determine the type of endpoint.
5. Install the Block I/O protocol.

19.3.2.3 Example implementing Driver Binding Stop()

The Driver Binding Protocol `Stop()` service performs the reverse steps of the `Start()` service. Continuing with the previous example, the `Stop()` service uninstalls the Block I/O Protocol and closes the USB I/O Protocol. It also frees various allocated resources such as the private data structure.

19.3.3 I/O Protocol Implementations

The following examples reference a private context data structure called `USB_MOUSE_DEV`. The example below shows the portion of this data structure required for the other examples.

```
#include <Uefi.h>
#include <Protocol/UsbIo.h>
#include <Protocol/SimplePointer.h>

#define USB_MOUSE_DEV_PRIVATE_DATA_SIGNATURE SIGNATURE_32('U','s','b','M')

typedef struct {
    UINTN                         Signature;
    EFI_USB_IO_PROTOCOL            *UsbIo;
    EFI_SIMPLE_POINTER_PROTOCOL    SimplePointer;
    EFI_SIMPLE_POINTER_STATE       State;
    EFI_USB_ENDPOINT_DESCRIPTOR   IntEndpointDescriptor;
    BOOLEAN                        StateChanged;
} USB_MOUSE_DEV;

#define USB_MOUSE_DEV_FROM_MOUSE_PROTOCOL(a) \
    CR(a, USB_MOUSE_DEV, SimplePointer, USB_MOUSE_DEV_PRIVATE_DATA_SIGNATURE)
```

Example 208—USB Mouse Private Context Data Structure

This example uses the USB mouse driver to shows how the USB device driver can setup asynchronous interrupt transfers from the Driver Binding Protocol `Start()` service.

```
#include <Uefi.h>
#include <Protocol/UsbIo.h>

Status = UsbIo->UsbAsyncInterruptTransfer (
    UsbIo,
    EndpointAddr,
    TRUE,
    PollingInterval,
    PacketSize,
    OnMouseInterruptComplete,
    UsbMouseDevice
);
```

Example 209—Setup asynchronous interrupt transfer for USB mouse driver

The next example shows the corresponding asynchronous interrupt transfer callback function called `OnMouseInterruptComplete()`. In this function, if the passing `Result` parameter indicates an error, it clears the endpoint error status, unregisters the previous asynchronous interrupt transfer, and initiates another asynchronous interrupt transfer. If there is no error, it set the mouse state change indicator to `TRUE` and put the data that is read into the appropriate data structure.

```

#include <Uefi.h>
#include <Protocol/UsbIo.h>
#include <Library/UefiUsbLib.h>

EFI_STATUS
EFIAPI
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 210—Completing an asynchronous interrupt transfer

This example shows the `GetMouseState()` service of the Simple Pointer Protocol that the USB mouse driver produces. `GetMouseState()` does not initiate any asynchronous interrupt transfer requests. It simply checks the mouse state change indicator. If there is mouse input, it copies the mouse input to the passing `MouseState` data structure.

```
#include <Uefi.h>
#include <Protocol/UsbIo.h>
#include <Protocol/SimplePointer.h>
#include <Library/BaseMemoryLib.h>

EFI_STATUS
EFIAPI
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;
    }

    CopyMem (MouseState, &MouseDev->State, sizeof(EFI_SIMPLE_POINTER_STATE));

    //
    // Clear previous move state
    //
    // ...
    //

    return EFI_SUCCESS;
}
```

Example 211—Retrieving pointer movement

19.3.4 State machine consideration

To implement USB device support, the USB device drivers must maintain a state machine for their own transaction process. For example, the USB mass storage driver must 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 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 is no data available from the device. It timeouts this data-phase operation.
- The state machine is then in the status phase. It asks 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 resets the device and retries 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 becomes 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.

TIP: Make sure USB device drivers maintain a state machine for their own transaction process. The driver might need to retry transactions in the data phase in order to avoid dead loops and other errors.

19.4 Debug Techniques

Several techniques can be used to debug the USB driver stack. The following discussions describe these techniques.

19.4.1 Debug Message Output

One typical debug technique is to output debug messages. The EDK II library [DebugLib](#) provides the `DEBUG()` and `ASSERT()` macros to output debug messages (see [Chapter 31](#) of this guide for details on the usage of the `DEBUG()` and `ASSERT()` macros). Messages may be sent at the entry point and exit point of functions. When this is done, a log of the call stack is produced that may help locate the source of the error. It is not suggested to print the debug message in a frequently called function, such as a timer handler because this can starve execution cycles at lower TPLs and can significantly change the behavior of the drivers under debug.

19.4.2 USB Bus Analyzer

There are still some conditions that the `DEBUG()` and `ASSERT()` macros are not sufficient for a developer to find the problem. One way to gain more debug information is to use a USB bus analyzer. Because a bus analyzer is inserted between the host and the device, the bus analyzer can monitor all the traffic on a single USB cable. Having access to the USB bus traffic information can make it easier to root cause some difficult bugs—for example, when a host controller loses packets on some occasions. Also, for the state machine chaos problem that was introduced in [Section 19.3.4](#), a bus analyzer can display the packet sequences and the unfinished state machine. This can help quickly solve that type of problem.

19.4.3 USBCheck/USBCV Tool

Another useful tool for debugging is the USBCheck/USBCV tool from <http://www.usb.org>. This tool is very helpful in determining if a device complies with a specific driver. 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.

19.5 Nonconforming USB Devices

There are debates on how best to handle devices that do not conform to the *USB Specification*. It is recommended that the driver stack comply with the *USB Specification* and reject any nonconforming devices. A nonconforming device that is not linked into the USB software stack should not interact further with the system.

However, even if the device is nonconforming and the USB driver stack should reject it, developers need to make sure that the nonconforming device do not cause system failures. The developer must not make any assumptions about the device's behavior, especially since, once a system is known not to conform, its behavior cannot be trusted. It can respond to addressing that was not meant for that device; it can corrupt data going into it and coming back from it; and it cannot be trusted to perform its intended function(s). It is essential for the end-user's experience that the nonconforming device does not negatively affect the system.

A driver can only reliably reject nonconforming devices that it already knows about. For USB devices, the identity of devices may be determined by use of the data in the USB device description packets.

USB devices have several sets of known issues that may be detected and hidden from the user. For example, some keyboards auto-repeat when keys are pressed for an extended period of time. In this case the consuming driver should simply ignore packets which repeatedly provide identical information. Media devices also have several issues. USB requires implementation of the SCSI or ATAPI specifications, which, for many e.g. thumb drives, is beyond their capacity. As such, relying only on basic commands can greatly increase the probability of functionality.

20

SCSI Driver Design Guidelines

There are several categories of SCSI drivers that cooperate to provide the SCSI driver stack in a platform. Table 31 lists these SCSI drivers.

Table 31—Classes of SCSI drivers

Class of driver	Description
SCSI host controller driver	Consumes PCI I/O Protocol on the SCSI host controller handle and produces the Ext SCSI Pass Thru Protocol. If a driver is required to be compatible with the EFI 1.10 Specification, then the SCSI Pass Thru Protocol must be produced.
SCSI bus driver	Consumes the Ext SCSI Pass Thru Protocol and produces a child handle for SCSI targets on the SCSI bus. Installs the Device Path Protocol and SCSI I/O Protocol onto each child handle.
SCSI device driver	Consumes the SCSI 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-conformant operating system.

This chapter shows how to write UEFI Drivers for SCSI host controllers and UEFI Drivers for SCSI devices. SCSI drivers must follow all of the general design guidelines described in [Chapter 4](#) of this guide. In addition, any SCSI host controllers that are PCI controllers must also follow the PCI-specific design guidelines described in [Chapter 18](#). This chapter covers the guidelines that apply specifically to the management of SCSI host controllers, SCSI channels, and SCSI devices. SCSI drivers, especially those for RAID controllers, may include HII functionality for SCSI subsystem configuration settings. HII functionality is described in [Chapter 12](#) of this guide.

The *EFI 1.10 Specification* defines the SCSI Pass Thru Protocol. UEFI Drivers for SCSI host controllers that are required to work properly on platforms that conform to the *EFI 1.10 Specification* are required to produce the SCSI Pass Thru Protocol and also produce the Block I/O protocol for physical and logical drives that the SCSI host controller manages. This implies that a UEFI Driver for the SCSI host controller in an EFI 1.10 platform is required to perform all the functions of the SCSI driver stack described in the [table above](#). The *UEFI 2.0 Specification* and above require the platform firmware to provide the SCSI bus driver and SCSI device driver for mass storage devices, so the implementation of a UEFI Driver for a SCSI host controller is simpler if the UEFI Driver is only required to function properly on platforms that conform to the *UEFI 2.0 Specification* and above.

20.1 SCSI Host Controller Driver

A SCSI host controller driver manages a SCSI host controller that contains one or more SCSI channels. It creates handles for each SCSI channel and installs the Extended

SCSI Pass Thru Protocol and Device Path Protocol to each of the handle that the driver creates. See the SCSI Driver Models and Bus Support chapter of the *UEFI Specification* for details about [EFI_EXT_SCSI_PASS_THRU_PROTOCOL](#).

A SCSI host controller driver follows the UEFI driver model. Depending on the adapter that it manages, a SCSI host controller driver can be categorized as either a device driver or a hybrid driver. It creates child handles for each SCSI channel (if there is more than 1) and it may also install protocols on its own handle. Typically, SCSI host controller drivers are chip-specific because of the requirement to initialize and manage the currently bound SCSI host controller.

Because there may be multiple SCSI host adapters in a platform that may be managed by a single SCSI host controller driver, it is recommended that the SCSI host controller driver be designed to be re-entrant and allocate a different private context data structure for each SCSI host controller.

20.1.1 Single-Channel SCSI Adapters

If the SCSI adapter supports one channel, then the SCSI host controller driver performs the following:

- Install Extended SCSI Pass Thru Protocol onto the controller handle for the SCSI host controller.
- Set the logical attribute for the SCSI channel in the mode structure.
- Set the physical attribute for the SCSI channel in the mode structure.

The [following figure](#) shows an example implementation on a single-channel SCSI adapter. The green layer represents the SCSI host controller driver.

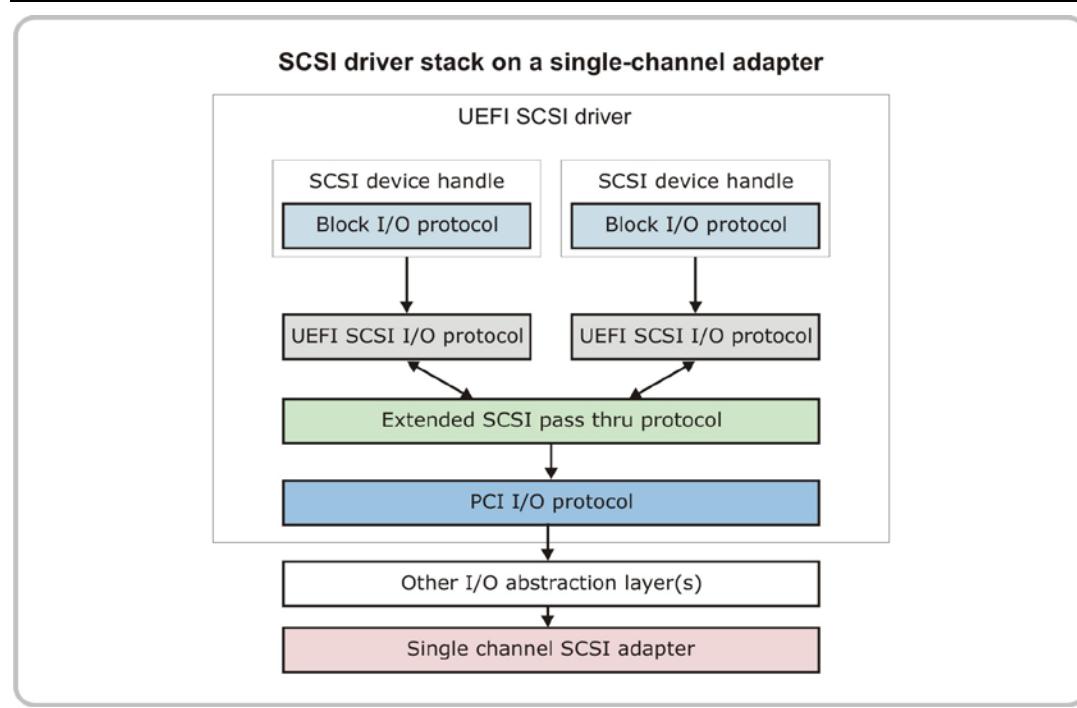


Figure 22—Sample SCSI driver stack on single-channel adapter

Because there is only one SCSI channel, the SCSI driver can simply implement one instance of the Extended SCSI Pass Thru Protocol. The platform firmware provides the SCSI Bus Driver and SCSI Disk Driver that complete the driver stack by performing the following actions:

- Scan for SCSI targets on the SCSI channel and create child handles.
- Install Device Path Protocol to each child handle.
- Install SCSI I/O Protocol to each child handle.
- Install I/O abstraction such as the Block I/O Protocol to each child handle.

20.1.2 Multi-Channel SCSI Adapters

A SCSI host controller driver is more complex if the SCSI adapter provides multiple SCSI channels. The [following figure](#) shows a possible SCSI driver implementation on a two-channel SCSI adapter.

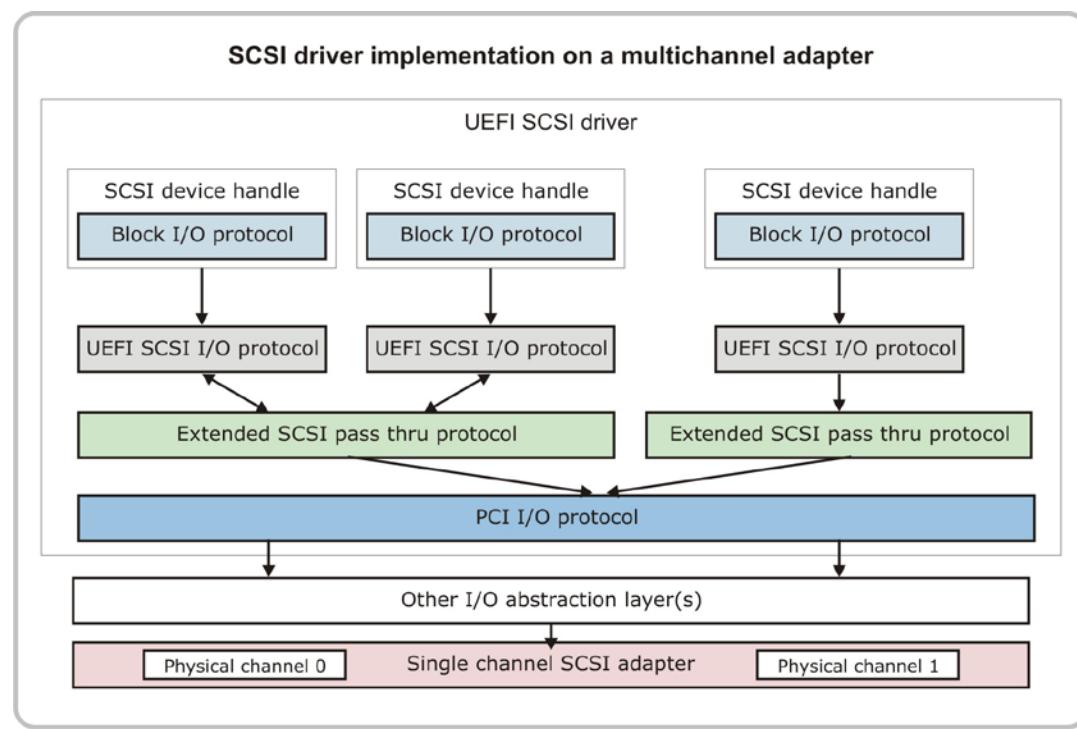


Figure 23—Sample SCSI driver implementation on a multichannel adapter

In this case, the SCSI adapter produces two physical SCSI channels by performing the following:

- Create a child handle for each physical SCSI channel.
- Install Device Path Protocol to each child handle.
- Install Extended SCSI Pass Thru Protocol onto each child handle
- Set the logical attribute for the SCSI channel in the mode structure on each child handle.
- Set the physical attribute for the SCSI channel in the mode structure on each child handle.

The platform firmware provides the SCSI Bus Driver and SCSI Disk Driver that complete the two driver stacks on each of the Extended SCSI Pass Thru Protocols shown above by performing the following actions:

- Scan for SCSI targets on each SCSI channel and create child handles.
- Install Device Path Protocol to each child handle.
- Install SCSI I/O Protocol to each child handle.
- Install I/O abstraction such as the Block I/O Protocol to each child handle.

20.1.3 SCSI Adapters with RAID

A SCSI host controller driver may also support SCSI adapters with RAID capability. The [following figure](#) shows an example implementation with two physical SCSI channels and one logical channel. The 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 Extended SCSI Pass Thru installed, but these are not be used except for diagnostic use. For the logical channel, the SCSI host controller driver produces another Extended SCSI Pass Thru Protocol (with physical bit turned off) instance based on the RAID configuration. Requests sent to the Extended SCSI Pass Thru protocol for the logical channel are processed by the SCSI host controller drivers and converted into requests on the physical SCSI channels. The platform firmware must only enumerate and boot from SCSI targets present on the logical SCSI channel.

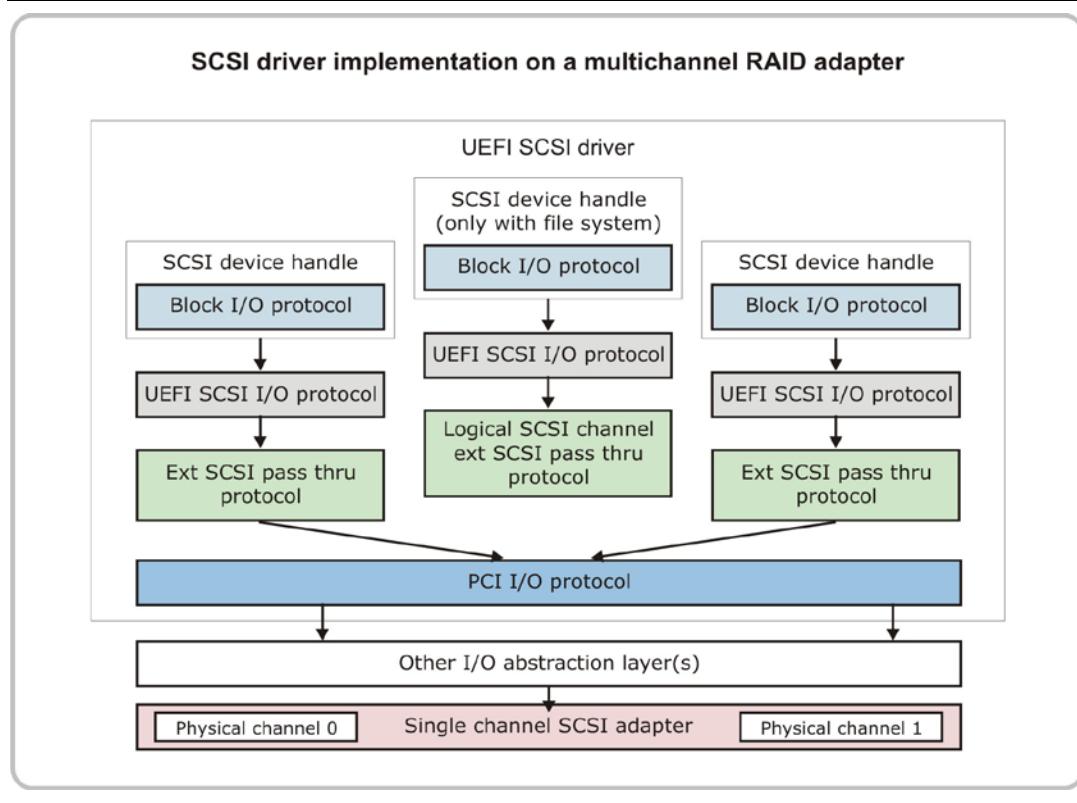


Figure 24—Sample SCSI driver implementation on multichannel RAID adapter

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 host controller driver is only required to produce a single logical channel for the RAID.

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 Extended 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 SCSI host controller driver may be designed and implemented for a wide variety of SCSI adapters types, and those SCSI host controller drivers can produce the Extended SCSI Pass Thru Protocol for SCSI channels that contain SCSI targets that may be used as UEFI boot devices.

20.1.4 Implementing driver binding protocol

A SCSI host controller driver follows the UEFI driver model, so the image entry point of a SCSI host controller 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 a SCSI host controller driver.

20.1.4.1 Supported()

The `Supported()` function tests to see if a given controller handle is SCSI adapter the driver knows how to manage. In this function, a SCSI host controller driver checks to see if the `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 host controller driver checks the `ClassCode`, `VendorId`, and `DeviceId` from the device's PCI configuration header to see if it is a conformant SCSI adapter that can be managed by the SCSI host controller driver.

20.1.4.2 Start()

The `start()` function tells the SCSI host controller driver to start managing the SCSI host controller. In this function, a single channel SCSI host controller driver uses chip-specific knowledge to perform the following tasks:

- Initialize the SCSI host controller.
- Enable the PCI device.
- Allocate resources.
- Construct data structures for the driver to use.
- Install the Extended SCSI Pass Thru Protocol instance on the same handle that has the PCI I/O Protocol.

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 SCSI host controller.
- Create child handles for each physical SCSI channel.
- Append the device path for each channel handle.
- Install the Device Path Protocol and Extended SCSI Pass Thru Protocol on every newly created channel handle.

20.1.4.3 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 a SCSI host controller 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 Extended SCSI Pass Thru abstraction layer.

20.1.5 Implementing Extended SCSI Pass Thru 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 Extended SCSI Pass Thru Protocol is usually implemented in the file `ExtScsiPassThru.c`. [Appendix A](#) contains a template for the Extended SCSI Pass Thru Protocol.

```
typedef struct _EFI_EXT_SCSI_PASS_THRU_PROTOCOL EFI_EXT_SCSI_PASS_THRU_PROTOCOL;
{
    /**
     * The EFI_EXT_SCSI_PASS_THRU_PROTOCOL provides information about a SCSI channel
     * and the ability to send SCI Request Packets to any SCSI device attached to
     * that SCSI channel. The information includes the Target ID of the host
     * controller on the SCSI channel and the attributes of the SCSI channel.
     */
    struct _EFI_EXT_SCSI_PASS_THRU_PROTOCOL {
        /**
         * A pointer to the EFI_EXT_SCSI_PASS_THRU_MODE data for this SCSI channel.
         */
        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_PATH BuildDevicePath;
        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;
    };
};
```

Example 212—Extended SCSI Pass Thru Protocol

For a detailed description of `EFI_EXT_SCSI_PASS_THRU_PROTOCOL`, see the section in the *UEFI Specification* on SCSI Driver Models and Bus Support.

Before implementing Extended SCSI Pass Thru Protocol, the SCSI host controller driver configures the SCSI host controller to a defined state. In practice, the SCSI adapter maps a set of SCSI host controller 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 host controller driver uses 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 must be completed before any Extended SCSI Pass Thru Protocol functions are called. The initialization is usually performed in the Driver Binding Protocol's `Start()` function of the SCSI host controller driver prior to installing the Extended SCSI Pass Thru Protocol instance into the Handle Database.

`EFI_EXT_SCSI_PASS_THRU_PROTOCOL.Mode` is a structure that describes the intrinsic attributes of Extended SCSI Pass Thru Protocol instance. Note that a non-RAID SCSI channel sets both the physical and logical attributes. A physical channel on the RAID adapter only sets the physical attribute, and the logical channel on the RAID adapter only sets the logical attribute. If the channel supports non-blocking I/O, the non-blocking attribute is also set. The example below shows how to set those attributes on a non-RAID SCSI adapter that supports non-blocking I/O.

```
//  
// Target Channel Id  
//  
ExtScsiPassThruMode.AdapterId = 4;  
//  
// The channel does support nonblocking I/O  
//  
ExtScsiPassThruMode.Attributes = EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL |  
                                 EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL |  
                                 EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO;  
//  
// Do not have any alignment requirement  
//  
ExtScsiPassThruMode.IoAlign = 0;
```

Example 213—SCSI Pass Thru Mode Structure for Single Channel Adapter

Example 214 shows how to set the SCSI `Mode` structure on a multi-channel non-RAID adapter. The example fits for either channel in [Figure 23](#).

```
//  
// Target Channel Id  
//  
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;  
//  
// Data must be aligned on a 4-byte boundary  
//  
ExtScsiPassThruMode.IoAlign = 2;
```

Example 214—SCSI Pass Thru Mode Structure for Multi-Channel Adapter

The next example shows how to set the corresponding *Mode* structures for both the physical and logical channel to be filled as shown below.

```
// ..... 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 215—SCSI Pass Thru Mode Structures for RAID SCSI adapter

The `EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetNextTarget()` and `EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetTargetLun()` functions provide the ability to enumerate the SCSI targets attached to a SCSI channel. The SCSI host controller driver may implement it by internally maintaining active device flags. The SCSI host controller driver may use this flag and channel-specific knowledge to determine 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 Extended SCSI Pass Thru Protocol may be used to abstract access to many different types of device, and as a result the specific device path used to describe a SCSI target may vary. The detailed SCSI target category can be identified only by the Extended SCSI Pass Thru implementation, which is why this function is part of the Extended SCSI Pass Thru Protocol.

The `EFI_EXT_SCSI_PASS_THRU_PROTOCOL.PassThru()` function is the most important function when implementing Extended SCSI Pass Thru Protocol and it performs 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 blocking and non-blocking data transfers, some may only support blocking transfers. In this case, the SCSI driver may implement the blocking SCSI I/O that is required by the *UEFI 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.

20.1.6 SCSI command set device considerations

Extended 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 mass storage devices and allows a 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 Extended SCSI Pass Thru Protocol. The implementation of this protocol also enables additional functionality in the future without modifying the SCSI drivers that are built on top of the SCSI host controller driver. Furthermore, Extended 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 Extended SCSI Pass Thru Protocol on SCSI command set-compatible technology.

20.1.6.1 ATAPI

This section provides guidance on how to implement the Extended SCSI Pass Thru Protocol for ATAPI devices.

Decoding the Target and Lun pair uses the intrinsic property of the technology or device. For ATAPI, only four devices are supported, 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. The example below shows how to build a device path node for an ATAPI device.

```
#include <Uefi.h>
#include <Protocol/ScsiPassThruExt.h>
#include <Protocol/DevicePath.h>
#include <Library/DevicePathLib.h>

EFI_STATUS
EFIAPI
AbcBuildDevicePath (
    IN     EFI_EXT_SCSI_PASS_THRU_PROTOCOL *This,
    IN     UINT8                          *Target,
    IN     UINT64                         Lun,
    IN OUT EFI_DEVICE_PATH_PROTOCOL      **DevicePath
)
{
    ATAPI_DEVICE_PATH *Node;

    Node = (ATAPI_DEVICE_PATH *)CreateDeviceNode (
        MESSAGING_DEVICE_PATH,
        MSG_ATAPI_DP,
        sizeof (ATAPI_DEVICE_PATH)
    );
    if (Node == NULL) {
        return EFI_OUT_OF_RESOURCES;
    }
}
```

```

    }

    Node->PrimarySecondary = (UINT8)(*Target >> 1);
    Node->SlaveMaster      = (UINT8)(*Target & 0x01);
    Node->Lun               = (UINT16)Lun;

    *DevicePath = (EFI_DEVICE_PATH_PROTOCOL *)Node;

    return EFI_SUCCESS;
}

```

Example 216—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` services can be found in the EDK II `MdeModulePkg` in the directory `MdeModulePkg\Bus\Ata\AtaAtapiPassThru`.

For the non-blocking I/O `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` function, the SCSI driver submits the SCSI command and returns. 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 UEFI firmware then schedules the notification function of the client event to be called.

The following example shows a sample non-blocking Extended SCSI Pass Thru Protocol implementation.

```

#include <Uefi.h>
#include <Protocol/ScsiPassThruExt.h>
#include <Library/UefiBootServicesTableLib.h>
#include <Library/UefiLib.h>

#define ATAPI_SCSI_PASS_THRU_DEV_SIGNATURE SIGNATURE_32('A','t','a','S')

typedef struct {
    UINTN                         Signature;
    EFI_HANDLE                     Handle;
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL ScsiPassThru;
    EFI_EXT_SCSI_PASS_THRU_MODE    ScsiPassThruMode;
    EFI_EVENT                      ClientEvent;
} ATAPI_SCSI_PASS_THRU_DEV;

#define ATAPI_SCSI_PASS_THRU_DEV_FROM_THIS(a) \
    CR(a, ATAPI_SCSI_PASS_THRU_DEV, ScsiPassThru, ATAPI_SCSI_PASS_THRU_DEV_SIGNATURE)

VOID
EFIAPI
AbcScsiPassThruPollEventNotify (
    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 (AtapiScsiPrivate->ClientEvent);
}
}

EFI_STATUS
EFIAPI
AbcScsiPassThru (
    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 (
        EVT_TIMER | EVT_NOTIFY_SIGNAL,
        TPL_CALLBACK,
        AbcScsiPassThruPollEventNotify,
        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,
        EFI_TIMER_PERIOD_MILLISECONDS (200)
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    //

```

```
// Submit SCSI I/O command through IDE I/O registers and return
//
// ...
//

return Status;
}
```

Example 217—Non-Blocking Extended SCSI Pass Thru Protocol Implementation

20.1.7 Discover a SCSI channel

It is recommended that the SCSI host controller driver construct a private context structure for each enumerated SCSI channel. See [Chapter 8](#) in this guide for the advantage of using such a private context structure.

Specifically, the SCSI host controller driver should store all required information for the child SCSI channel in this data structure, this should include 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 is described in [Chapter 8](#) of this document.

The method for determining the number of channels on a given controller is chip 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 Extended SCSI Pass Thru Protocol and Device Path Protocol on the appropriate handle (child handle is optional for single channel).

20.1.8 SCSI Device Path

The SCSI host controller driver described in this document 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.

20.1.8.1 SCSI Device Path Example

The [table below](#) shows an example device path for a SCSI host controller that supports a single SCSI channel and is located at PCI device number 0x07 and PCI function 0x00. The PCI SCSI host controller 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 32—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

20.1.8.2 Multiple SCSI channels on a multifunction PCI controller

A SCSI host controller with multiple SCSI channels on a multi-function PCI controller only changes the PCI portion of the device path for each SCSI channel. In this example, 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

20.1.8.3 Multiple SCSI channels on a single function PCI controller

If there is a SCSI PCI controller with multiple SCSI channels connected to a single function PCI device the device paths must differentiate the SCSI channels. In this example, SCSI channel 0 is accessed through Controller #0 below PCI function #0, and SCSI channel 1 is accessed through Controller #1 below PCI function #1. The following are the device paths for these SCSI channels:

- **ACPI(PNP0A03,1)/PCI(7|0)/Controller(0)** Access to channel 0
- **ACPI(PNP0A03,1)/PCI(7|0)/Controller(1)** Access to channel 1

20.1.9 Using Extended 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.

The following example demonstrates how to use Extended SCSI Pass Thru Protocol to perform blocking and non-blocking I/O.

```
#include <Uefi.h>
#include <Protocol/ScsiPassThruExt.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS
EFIAPI
ScsiPassThruTests (
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL *ScsiPassThru,
    UINT8                           *Target,
    UINT64                          Lun
)
{
    EFI_STATUS                      Status;
    EFI_EXT_SCSI_PASS_THRU_SCSI_REQUEST_PACKET  Packet;
    EFI_EVENT                        Event;

    //
    // Fill in Packet for the requested test operation
    //

    //
    // Blocking I/O
    //
    Status = ScsiPassThru->PassThru (
        ScsiPassThru,
        Target,
        Lun,
        &Packet,
        NULL
    );

    //
    // Non Blocking I/O
    //
    Status = gBS->CreateEvent (
        EVT_NOTIFY_SIGNAL,
        TPL_CALLBACK,
        NULL,
        NULL,
        &Event
    );

    Status = ScsiPassThru->PassThru (
        ScsiPassThru,
        Target,
        Lun,
        &Packet,
        &Event
    );

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

return Status;
}
```

Example 218—Blocking and non-blocking modes

20.2 SCSI Bus Driver

EDK II contains a generic SCSI bus driver. This driver uses the services of `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` to enumerate SCSI devices and produce child handles with `EFI_DEVICE_PATH_PROTOCOL` and `EFI_SCSI_IO_PROTOCOL`. The implementation of the SCSI Bus Driver is found in the `MdeModulePkg` in the directory `MdeModulePkg/Bus/Scsi/ScsiBusDxe`.

If UEFI-based system firmware is ported to a new platform, most of the SCSI-related changes occur in the implementation of the SCSI host controller driver. If new types of SCSI devices are introduced that are required to provide a console or provide a UEFI boot capability, then the implementation of new SCSI Device Drivers are also required. The SCSI bus driver is designed to be a generic, platform-agnostic driver. As a result, customizing the SCSI bus driver is **strongly discouraged**. The detailed design and implementation of the SCSI bus driver is not covered in this guide.

20.3 SCSI Device Driver

SCSI device drivers use services provided by `EFI_SCSI_IO_PROTOCOL` to produce one or more protocols that provide I/O abstractions of a SCSI device. SCSI device drivers must follow the UEFI Driver Model. The EDK II provides a SCSI Device Driver for block-oriented SCSI devices such as hard drives, CD-ROM, and DVD-ROM. The implementation of the SCSI Disk Driver is found in the `MdeModulePkg` in the directory `MdeModulePkg/Bus/Scsi/ScsiDiskDxe`.

20.3.1 Driver Binding Protocol Supported()

SCSI device drivers must implement the `EFI_DRIVER_BINDING_PROTOCOL` that contains the `Supported()`, `Start()`, and `Stop()` services. The `Supported()` service checks the controller handle that has been passed in to see whether this handle represents a SCSI 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_SCSI_IO_PROTOCOL` installed. If not, this handle is not a SCSI device on the current SCSI channel.
- Retrieve the 8-bit SCSI device type to see if the type is one that this driver can manage.

If the above two checks are passed, it means that the SCSI device driver can manage the device that the controller handle represents. The `Supported()` service returns `EFI_SUCCESS`. Otherwise, the `Supported()` service returns `EFI_UNSUPPORTED`. In addition, this check process must not disturb the current state of the SCSI device, because another SCSI device driver may be managing this SCSI device.

The [following example](#) shows an implementation of the Driver Binding Protocol `Supported()` service for a SCSI mass storage device. It opens the SCSI I/O Protocol with an attribute of `EFI_OPEN_PROTOCOL_BY_DRIVER`. It then used the `GetDeviceType()` service of the SCSI I/O Protocol and evaluates the type information to see if it is a hard drive or a CD-ROM.

```

#include <Uefi.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/ScsiIo.h>
#include <IndustryStandard/Scsi.h>
#include <Library/UefiBootServicesTableLib.h>

EFI_STATUS
EFIAPI
AbcSupported (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath OPTIONAL
)
{
    EFI_STATUS             Status;
    EFI_SCSI_IO_PROTOCOL *ScsiIo;
    UINT8                 DeviceType;

    //
    // Open the SCSI I/O Protocol on ControllerHandle
    //
    Status = gBS->OpenProtocol (
        ControllerHandle,
        &gEfiScsiIoProtocolGuid,
        (VOID **)&ScsiIo,
        This->DriverBindingHandle,
        ControllerHandle,
        EFI_OPEN_PROTOCOL_BY_DRIVER
    );
    if (EFI_ERROR (Status)) {
        return Status;
    }

    //
    // Get the SCSI Device Type
    //
    Status = ScsiIo->GetDeviceType (ScsiIo, &DeviceType);
    if (EFI_ERROR(Status)) {
        goto Done;
    }

    //
    // Check to see if the interface descriptor is supported by this driver
    //
    if ((DeviceType != EFI_SCSI_TYPE_DISK) &&
        (DeviceType != EFI_SCSI_TYPE_CDROM)) {
        Status = EFI_UNSUPPORTED;
    }

Done:
    //
    // Close the SCSI I/O Protocol
    //
    gBS->CloseProtocol (
        ControllerHandle,
        &gEfiScsiIoProtocolGuid,
        This->DriverBindingHandle,
        ControllerHandle
    );

    return Status;
}

```

Example 219—Supported() for a SCSI device driver

20.3.2 Driver Binding Protocol Start() and Stop()

The `Start()` service of the Driver Binding Protocol for a SCSI device driver or host controller driver opens the SCSI I/O Protocol with an attribute of `EFI_OPEN_PROTOCOL_BY_DRIVER`. The service then installs the I/O abstraction protocol for the SCSI device onto the handle on which the `EFI_SCSI_IO_PROTOCOL` is installed.

20.3.3 I/O Protocol Implementations

Once a SCSI device driver has been started, it must process I/O requests for the I/O abstraction that was installed in Driver Binding `Start()`. In the case of the SCSI Disk Driver, these I/O abstractions are the Block I/O Protocol, the Block I/O 2 Protocol, and optionally the Storage Security Command Protocol. The Block I/O Protocols use the SCSI I/O Protocol to build SCSI command to perform operations to detect drive capabilities, read sectors, and write sectors. The EDK II `MdePkg` also provide the library called `UefiScsiLib` that provides functions to simplify the use of the SCSI I/O Protocol.

ATA Driver Design Guidelines

There are several categories of ATA drivers that cooperate to provide the ATA driver stack in a platform. The following table lists these ATA drivers.

Table 33—Classes of ATA drivers

Class of driver	Description
ATA host controller driver	Consumes PCI I/O Protocol on the ATA host controller handle and produces the ATA Pass Thru Protocol used to access hard drives and the Ext SCSI Pass Thru Protocol used to access CD-ROM/DVD-ROM drives.
ATA bus driver	Consumes the ATA Pass Thru Protocol and creates child handles for ATA targets with the Device Path Protocol, Block I/O Protocol, Block I/O 2 Protocol, and optionally the Storage Security Command Protocol.

This chapter shows how to write UEFI Drivers for ATA host controllers. ATA drivers must follow all of the general design guidelines described in [Chapter 4](#) of this guide. In addition, any ATA host controllers that are PCI controllers must also follow the PCI-specific design guidelines described in [Chapter 18](#). This section covers the guidelines that apply specifically to the management of ATA host controllers. ATA drivers, especially those for RAID controllers, may include HII functionality for ATA subsystem configuration settings. HII functionality is described in [Chapter 12](#) of this guide.

21.1 ATA Host Controller Driver

An ATA host controller driver manages a ATA host controller and installs ATA Pass Thru Protocol and the Extended SCSI Pass Thru Protocol. See the ATA Pass Thru Protocol section of the *UEFI Specification* for details about [EFI_ATA_PASS_THRU_PROTOCOL](#) and [EFI_EXT_SCSI_PASS_THRU_PROTOCOL](#). Guidelines for the Extended SCSI Pass Thru Protocol are covered in [Chapter 20](#). The rest of this section focuses on the ATA Pass Thru Protocol.

An ATA host controller driver is a device driver that follows the UEFI driver model. Typically, ATA host controller drivers are chip-specific because of the requirement to initialize and manage the currently bound ATA host controller. Because there may be multiple ATA host adapters in a platform that may be managed by a single ATA host controller driver, it is recommended that the ATA host controller driver be designed to be re-entrant and allocate a different private context data structure for each ATA host controller.

An ATA host controller driver performs the following:

- Install the ATA Pass Thru Protocol onto the controller handle for the ATA host controller.

- Install Extended SCSI Pass Thru Protocol onto the controller handle for the SCSI host controller.

The platform firmware typically provides the ATA Bus Driver that completes the ATA driver stack by performing the following actions:

- Use the services of the ATA Pass Thru Protocol to scan for ATA targets connected to the ATA host controller and create child handles.
- Install Device Path Protocol to each child handle.
- Install Block I/O Protocol on each child handle.
- Install Block I/O 2 Protocol on each child handle.
- If the hard drive supports the SPC-4 or ATA8-ACS command set, then install the Storage Security Command Protocol the child handle.

21.1.1 Implementing Driver Binding Protocol

An ATA host controller driver follows the UEFI driver model, so the image entry point of a ATA host controller 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 a ATA host controller driver.

21.1.1.1 Supported()

The `Supported()` function tests to see if a given controller handle is an ATA controller the driver knows how to manage. In this function, an ATA host controller driver checks to see if the `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, an ATA host controller driver checks the `ClassCode`, `VendorId`, and `DeviceId` from the device's PCI configuration header to see if it is a conformant ATA controller that can be managed by the ATA host controller driver.

21.1.1.2 Start()

The `Start()` function tells the ATA host controller driver to start managing the ATA host controller. In this function, an ATA host controller driver uses chip-specific knowledge to perform the following tasks:

- Initialize the ATA host controller.
- Enable the PCI device.
- Allocate resources.
- Construct data structures for the driver to use.
- Install the ATA Pass Thru Protocol instance on the same handle that has the PCI I/O Protocol.
- Install the Extended SCSI Pass Thru Protocol instance on the same handle that has the PCI I/O Protocol.

If the ATA host controller provides RAID capabilities, then the ATA host controller driver can either choose to only expose access to the logical drives following the

algorithm above, or the ATA host controller driver can produce two instances of the ATA Pass Thru Protocol. One for accessing the physical drives, and another for accessing the logical drives. In this case, a child handle is created for each ATA Pass Thru Protocol instance.

21.1.1.3 Stop()

The `stop()` function performs the opposite operations as `start()`. Generally speaking, an ATA host controller driver is required to do the following:

- Disable the ATA controller.
- 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.

21.1.2 Implementing ATA Pass Thru Protocol

`EFI_ATA_PASS_THRU_PROTOCOL` allows information about a ATA target to be collected and allows ATA Request Packets to be sent to any ATA devices connected to the ATA host controller, even if those devices are not boot devices. This protocol is attached to the device handle of the ATA host controller 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 ATA hard drives allowing those devices to be used as boot devices. The ATA Pass Thru Protocol is usually implemented in the file `AtaPassThru.c`. [Appendix A](#) contains a template for the ATA Pass Thru Protocol.

```
typedef struct _EFI_ATA_PASS_THRU_PROTOCOL EFI_ATA_PASS_THRU_PROTOCOL;

struct _EFI_ATA_PASS_THRU_PROTOCOL {
    EFI_ATA_PASS_THRU_MODE           *Mode;
    EFI_ATA_PASS_THRU_PASSTHRU       PassThru;
    EFI_ATA_PASS_THRU_GET_NEXT_PORT  GetNextPort;
    EFI_ATA_PASS_THRU_GET_NEXT_DEVICE GetNextDevice;
    EFI_ATA_PASS_THRU_BUILD_DEVICE_PATH BuildDevicePath;
    EFI_ATA_PASS_THRU_GET_DEVICE     GetDevice;
    EFI_ATA_PASS_THRU_RESET_PORT    ResetPort;
    EFI_ATA_PASS_THRU_RESET_DEVICE  ResetDevice;
};
```

Example 220—ATA Pass Thru Protocol

For a detailed description of `EFI_ATA_PASS_THRU_PROTOCOL`, see the ATA Pass Thru Protocol section of the *UEFI Specification*.

Before implementing ATA Pass Thru Protocol, the ATA host controller driver configures the ATA host controller to a defined state. In practice, the ATA host controller maps a set of ATA host controller registers in I/O or memory-mapped I/O space. Although the detailed layout or functions of these registers vary from one ATA host controller to another, the ATA host controller driver uses specific knowledge to set up the proper ATA mode and configure the timing registers for the current mode. Other considerations include DMA engine and interrupt initialization, among others.

All the hardware-related settings must be completed before any ATA Pass Thru Protocol functions are called. The initialization is usually performed in the Driver Binding Protocol's `Start()` function of the ATA host controller driver prior to installing the ATA Pass Thru Protocol instance into the Handle Database.

`EFI_ATA_PASS_THRU_PROTOCOL.Mode` is a structure that describes the intrinsic attributes of the ATA Pass Thru Protocol instance. Note that a non-RAID ATA host controllers set both the physical and logical attributes. A physical channel on the RAID sets only the physical attribute, and the logical channel on the RAID adapter sets only the logical attribute. If the channel supports non-blocking I/O, the non-blocking attribute is also set. The example below shows how to set those attributes on a non-RAID ATA host controller that supports non-blocking I/O.

```
//  
// The channel does support nonblocking I/O  
//  
AtaPassThruMode.Attributes = EFI_ATA_PASS_THRU_ATTRIBUTES_PHYSICAL |  
                           EFI_ATA_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL |  
                           EFI_ATA_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO;  
  
//  
// Do not have any alignment requirement  
//  
AtaPassThruMode.IoAlign = 0;
```

Example 221—ATA Pass Thru Mode Structure

The example below shows how to set the ATA `Mode` structures for an ATA host controller that provides RAID capabilities and produced an ATA Pass Thru Protocol instance for accessing the physical drives and another ATA Pass Thru Protocol instance for accessing the logical drives.

```
//  
// ..... Physical Channel .....  
//  
AtaPassThruMode.Attributes = EFI_ATA_PASS_THRU_ATTRIBUTES_PHYSICAL |  
                           EFI_ATA_PASS_THRU_ATTRIBUTES_NONBLOCKIO;  
AtaPassThruMode.IoAlign = 0;  
  
//  
// ..... Logical Channel .....  
//  
AtaPassThruMode.Attributes = EFI_ATA_PASS_THRU_ATTRIBUTES_LOGICAL |  
                           EFI_ATA_PASS_THRU_ATTRIBUTES_NONBLOCKIO;  
AtaPassThruMode.IoAlign = 0;
```

Example 222—SCSI Pass Thru Mode Structures for RAID SCSI adapter

The `EFI_ATA_PASS_THRU_PROTOCOL.GetNextPort()` and `EFI_ATA_PASS_THRU_PROTOCOL.GetNextDevice()` functions provide the ability to enumerate all the ATA devices.

`EFI_ATA_PASS_THRU_PROTOCOL.BuildDevicePath()` function facilitates the construction of an ATA device path.

The `EFI_ATA_PASS_THRU_PROTOCOL.PassThru()` function is the most important function when implementing ATA Pass Thru Protocol and it performs the following:

- Initialize the internal register for command/data transfer.

- Put valid ATA commands 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 ATA host controllers support both blocking and non-blocking data transfers, some may only support blocking transfers.

21.1.3 ATA Command Set Considerations

ATA Pass Thru Protocol defines a method to directly access ATA devices. This protocol provides interfaces that allow a generic driver to produce the Block I/O Protocol for ATA devices and allows a UEFI utility to issue commands to any ATA device. The main reason to provide such an access is to enable S.M.A.R.T. functionality during POST. This enabling is accomplished using the generic interfaces that are defined in ATA Pass Thru Protocol. The implementation of this protocol also enables additional functionality in the future without modifying the ATA Bus Driver that is built on top of the ATA host controller driver.

21.1.4 ATA Device Paths

The table below shows an example device path for a ATA host controller that supports a single SCSI channel and is located at PCI device number 0x07 and PCI function 0x00. The PCI SCSI host controller 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 34—SATA device path examples

yte offset	yte length	Data	21.5 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

Byte offset	Byte length	Data	21.5 Description
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

21.6 ATA Bus Driver

EDK II contains a generic ATA bus driver. This driver uses the services of [EFI_ATA_PASS_THRU_PROTOCOL](#) to enumerate ATA devices and produce child handles with [EFI_DEVICE_PATH_PROTOCOL](#), [EFI_BLOCK_IO_PROTOCOL](#), [EFI_BLOCK_IO2_PROTOCOL](#), and optionally the [EFI_STORAGE_SECURITY_COMMAND_PROTOCOL](#). The implementation of the ATA Bus Driver is found in the [MdeModulePkg](#) in the directory [MdeModulePkg/Bus/Ata/AtaBusDxe](#).

If UEFI-based system firmware is ported to a new platform, most of the ATA-related changes occur in the implementation of the ATA host controller driver. The ATA bus driver is designed to be a generic, platform-agnostic driver. As a result, customizing the ATA bus driver is **strongly discouraged**. The detailed design and implementation of the ATA bus driver is not covered in this guide.

Text Console Driver Design Guidelines

This chapter covers the general guidelines for implementing UEFI Drivers for devices that provide console services. This include devices that allow the user to input information through key press actions such as a keyboard or keypad, devices that provide text based output, and byte-stream devices like a UART that can be connected to a remote terminal to provide console services.

If a device is intended to be used as a console input device and that device must be available for use as a console input device while UEFI firmware is active, then a UEFI Driver must be implemented that produces both the Simple Text Input Protocol and the Simple Text Input Ex Protocol. The Simple Text In Protocols are produced for any device that can behave like a basic keyboard. This could be an actual keyboard such as USB or PS/2, a serial terminal, a remote network terminal such as Telnet, or a custom device that provide the ability for a user to perform actions that can be translated into UEFI compatible keystroke information.

If a device is intended to be used as a console output device while UEFI firmware is active, and that device is able to display text strings, then a UEFI Driver must be implemented that produces the Simple Text Output Protocol. The device must support an 80 column by 25 row character mode, and may optionally support additional modes. The device must either directly support or be able to emulate the following operations:

- Clear the display
- Scroll the display up
- Move cursor
- Turn cursor on and off
- Support 16 foreground colors
- Support 8 background colors

If a device is graphics controller that is able to emulate a text console using bitmap fonts, then see [Chapter 23](#) on the Graphics Output Protocol. The EDK II provides a platform agnostic driver in the [**MdeModulePkg**](#) in the directory [**MdeModulePkg/Universal/Console/GraphicsConsoleDxe**](#) that uses the services of a Graphics Output Protocol and bitmap fonts to produce the Simple Text Output Protocol.

If a device supports character based communication where data can be both transmitted and received character at a time, and the goal is to use that device for console services by connecting the device to terminal or terminal emulator, then a UEFI Driver must be implemented that produces the Serial I/O Protocol. This may include devices such as a UART style serial port or any other character based I/O device on a motherboard, an add-in card, or USB.

The EDK II provides a terminal driver that supports the PC-ANSI, VT-100, VT-100+, and VT-UTF8 terminal types. This terminal driver is in the [**MdeModulePkg**](#) in the directory [**MdeModulePkg/Universal/Console/TerminalDxe**](#). This driver consumes the Serial I/O

Protocol and produces all the Simple Input Protocol, the Simple Input Ex Protocol, and the Simple Text Output Protocol.

22.1 Assumptions

The rest of this chapter assumes that the Driver Checklist in [Chapter 2](#) has been followed and that the following items have already been identified:

- UEFI Driver Type
- optional UEFI Driver features
- Supported CPU architectures
- Consumed protocols that are used to produce one or more of the console related protocols

UEFI drivers that produce console services typically follow the UEFI Driver Model because the devices are typically on industry standard busses such as PCI or USB. However, it is possible to implement UEFI drivers for console devices that are not on industry standard busses. In these cases a Root Bridge Driver implementation may be more appropriate than a UEFI Driver Model implementation.

22.2 Simple Text Input Protocol Implementation

The implementation of the Simple Text Input Protocols is typically found in the file `SimpleTextInput.c`. [Appendix A](#) contains a template for a `SimpleTextInput.c` file for a UEFI Driver. The list of tasks to implement the Simple Text Input Protocols is as follows:

- Add global variable for the `EFI_SIMPLE_TEXT_INPUT_PROTOCOL` instance to `SimpleTextInput.c`.
- Add global variable for the `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` instance to `SimpleTextInput.c`.
- Implement the `Reset()` and `ReadKeyStroke()` services in `SimpleTextInput.c` that is shared between the Simple Text Input Protocol and the Simple Text Input Ex Protocol.
- Implement the `SetState()`, `RegisterKeyNotify()` and `UnregisterKeyNotify()` services in `SimpleTextInput.c` for the Simple Text Input Ex Protocol.
- Implement notification function for the `WaitForKey` and `WaitForKeyEx` events in `SimpleTextInput.c` that is shared between the Simple Text Input Protocol and the Simple Text Input Ex Protocol.
- Create `WaitForKey` and `WaitForKeyEx` events before the Simple Input Protocols are installed into the Handle Database.
- If a device does not buffer keystrokes, or the buffer is very small, a timer event may be required to periodically read contents from a keyboard buffer.

[Example 223](#), following, shows the protocol interface structure for the Simple Text Input Protocol and [Example 224](#), below that, shows the protocol interface structure for the Simple Text Input Ex Protocol for reference. These two protocols are composed of services and each has an `EFI_EVENT` that may be used by the UEFI Boot Manager or

UEFI Applications to determine if a keystroke has been pressed. The UEFI Boot Services `WaitForEvent()` and `CheckEvent()` can be used to perform these checks on the events specified by `WaitForKey` and `WaitForKeyEx`.

```
typedef struct _EFI_SIMPLE_TEXT_INPUT_PROTOCOL  EFI_SIMPLE_TEXT_INPUT_PROTOCOL;
{
    /**
     * The EFI_SIMPLE_TEXT_INPUT_PROTOCOL is used on the ConsoleIn device.
     * It is the minimum required protocol for ConsoleIn.
     */
    struct _EFI_SIMPLE_TEXT_INPUT_PROTOCOL {
        EFI_INPUT_RESET      Reset;
        EFI_INPUT_READ_KEY   ReadKeyStroke;
        /**
         * Event to use with WaitForEvent() to wait for a key to be available
         */
        EFI_EVENT            WaitForKey;
    };
};
```

Example 223—Simple Text Input Protocol

```
typedef struct _EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL
    EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL;

{
    /**
     * The EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL is used on the ConsoleIn
     * device. It is an extension to the Simple Text Input protocol
     * which allows a variety of extended shift state information to be
     * returned.
     */
    struct _EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL{
        EFI_INPUT_RESET_EX      Reset;
        EFI_INPUT_READ_KEY_EX   ReadKeyStrokeEx;
        /**
         * Event to use with WaitForEvent() to wait for a key to be available.
         */
        EFI_EVENT                WaitForKeyEx;
        EFI_SET_STATE             SetState;
        EFI_REGISTER_KEYSTROKE_NOTIFY RegisterKeyNotify;
        EFI_UNREGISTER_KEYSTROKE_NOTIFY UnregisterKeyNotify;
    };
};
```

Example 224—Simple Text Input Ex Protocol

22.2.1 Reset() Implementation

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 dependent on the underlying hardware specifications. However, it is recommended that the basic functionality test perform as quickly as an operation as possible to support fast boot times.

22.2.2 ReadKeyStroke() and ReadKeyStrokeEx() Implementation

The `ReadKeyStroke()` and `ReadKeyStrokeEx()` functions are non-blocking operations that returns immediately with their `Key` and `KeyData` parameters containing the key code for the next key in the queue or it returns that there was no key code ready. These

functions never wait for a key to be pressed. `ReadKeyStroke()` may be implemented to layer on top of `ReadKeyStrokeEx()` to share as much logic as possible.

If a key is read, the device specific keystroke information, such as scan codes must be converted into `EFI_INPUT_KEY` and `EFI_KEY_DATA` structure contents. The Console Support chapter of the *UEFI Specification* provides the details on how different keys, toggle keys, and shift states are to be translated into these structures.

22.2.3 WaitForKey and WaitForKeyEx Notification Implementation

When the `WaitForKey` and `WaitForKeyEx` events are created, they must be associated with an event notification function. This event notification function checks to see if one or more keystrokes are currently available from the console input device. If one or more keystrokes are currently available from the console input device, then the `WaitForKey` and `WaitForKeyEx` events must be placed into the signaled state by calling the UEFI Boot Service `SignalEvent()`.

22.2.4 SetState() Implementation

The `SetState()` function sets the state on the input device such as Caps Lock, Num Lock, and Scroll Lock. Updating the state on the device being managed may perform actions such as changing the state of a user visible indicator, and also changes the keystroke information returned by `ReadKeyStroke()` and `ReadKeyStrokeEx()` for keys that are affected by state changes.

22.2.5 RegisterKeyNotify() Implementation

This function registers a notification function that is called when a specified keystroke is pressed by the user. This function must create a unique handle value that is returned, so a previous key registration can be unregistered using `UnregisterKeyNotify()`. The UEFI Driver is responsible for generating unique handle values so no two active registrations ever use the same handle value.

22.2.6 UnregisterKeyNotify() Implementation

This function unregisters a keystroke notification that was registered through `RegisterKeyNotify()`.

22.3 Simple Text Output Protocol Implementation

The implementation of the Simple Text Output Protocol is typically found in the file `SimpleTextOutput.c`. [Appendix A](#) contains a template for a `SimpleTextOutput.c` file for a UEFI Driver. The list of tasks to implement the Simple Text Output Protocol is as follows:

- Add global variable for the `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` instance to `SimpleTextOutput.c`.

- Add global variable for the `EFI_SIMPLE_TEXT_OUTPUT_MODE` structure to `SimpleTextOutput.c`.
- Implement the Simple Text Output Protocol services in `SimpleTextInput.c`.

The example below shows the protocol interface structure for the Simple Text Output Protocol for reference. This protocol is composed of nine services and a pointer to a `Mode` structure.

```
typedef struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL;

///
/// The SIMPLE_TEXT_OUTPUT protocol is used to control text-based output devices.
/// It is the minimum required protocol for any handle supplied as the ConsoleOut
/// or StandardError device. In addition, the minimum supported text mode of such
/// devices is at least 80 x 25 characters.
///
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;

    ///
    /// Pointer to SIMPLE_TEXT_OUTPUT_MODE data.
    ///
    EFI_SIMPLE_TEXT_OUTPUT_MODE *Mode;
};
```

Example 225—Simple Text Output Protocol

22.3.1 Reset() Implementation

The reset here can be as simple as resetting the mode and clearing the screen, as demonstrated by the following example. (The example is from the terminal driver located at `\Sample\Universal\Console\Terminal\DXe` in EDK II.)

```
Status = This->SetAttribute (
    This,
    EFI_TEXT_ATTR (This->Mode->Attribute & 0x0F, EFI_BACKGROUND_BLACK)
);
Status = This->SetMode (This, 0);
```

Example 226—Light reset of terminal driver

A reset can also easily perform more actions, as shown in the [following example](#). When the `ExtendedVerification` parameter is `TRUE` this same driver also resets the serial protocol that it is running on top of.

```

if (ExtendedVerification) {
    Status = SerialIo->Reset (SerialIo);
    if (EFI_ERROR (Status)) {
        return Status;
    }
}

```

Example 227—Full reset of terminal driver

22.3.2 `OutputString()` Implementation

`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. Check each character to see if it is printable as text or graphics. Characters that are not printable as text or graphics are skipped.
3. Print all printable characters.
4. Update position of the cursor in the *Mode.
5. Return `EFI_SUCCESS` or `EFI_WARN_UNKNOWN_GLYPH` if some had to be fixed before printing.

22.3.3 `TestString()` Implementation

The `TestString()` function verifies that all the characters in the string can be printed. That is why they do 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.

22.3.4 `QueryMode()` Implementation

The `QueryMode()` function returns information supported modes. The UEFI Driver is required to return the number of `Rows` and number of `Columns` for each supported `ModeNumber`. `ModeNumber` must be less than `Mode->MaxMode`.

Note: All devices that support the Simple Text Output Protocol must minimally support an 80 x 25 character mode. Additional modes are optional. This means a basic Simple Text Output Protocol implementation supports a single `ModeNumber` of 0 with a geometry of 80 `Columns` and 25 `Rows`, and reports a `Mode->MaxMode` value of 1.

The `QueryMode()` function is typically used one of two ways:

1. Query for the geometry of the current mode. The [following line](#) populates the `Columns` and `Rows` variables with the geometry of the currently active console output.

```
#include <Uefi.h>
#include <Protocol/SimpleTextOut.h>

EFI_STATUS Status;
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *SimpleTextOutput;
UINTN Columns;
UINTN Rows;

Status = SimpleTextOutput->QueryMode (
    SimpleTextOutput,
    SimpleTextOutput->Mode->Mode,
    &Columns,
    &Rows
);
```

Example 228—Query current Simple Text Output Mode

2. Loop through all valid geometries that a given console can support. The following line populates (repeatedly) the *Column* and *Row* variables with the geometry of the each supported output mode.

```
#include <Uefi.h>
#include <Protocol/SimpleTextOut.h>

EFI_STATUS Status;
UINTN Index;
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *SimpleTextOutput;
UINTN Columns;
UINTN Rows;

for (Index = 0 ; Index < SimpleTextOutput->Mode->MaxMode ; Index++) {
    Status = SimpleTextOutput->QueryMode (
        SimpleTextOutput,
        Index,
        &Columns,
        &Rows
    );
}
```

Example 229—Query all Simple Text Output Modes

22.3.5 SetMode() Implementation

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.

Note: All devices that support the Simple Text Output Protocol must minimally support an 80 x 25 character mode. Additional modes are optional.

22.3.6 SetAttribute() Implementation

Setting the attributes is how the upper layers define how the screen printing should occur. This affects the background and foreground colors that are used when either

`OutputString()` or `ClearScreen()` is called. This function by itself does not change anything already printed to the console.

22.3.7 `ClearScreen()` Implementation

`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).

22.3.8 `SetCursorPosition()` Implementation

`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()`. The following figure shows a representation of the screen coordinates.

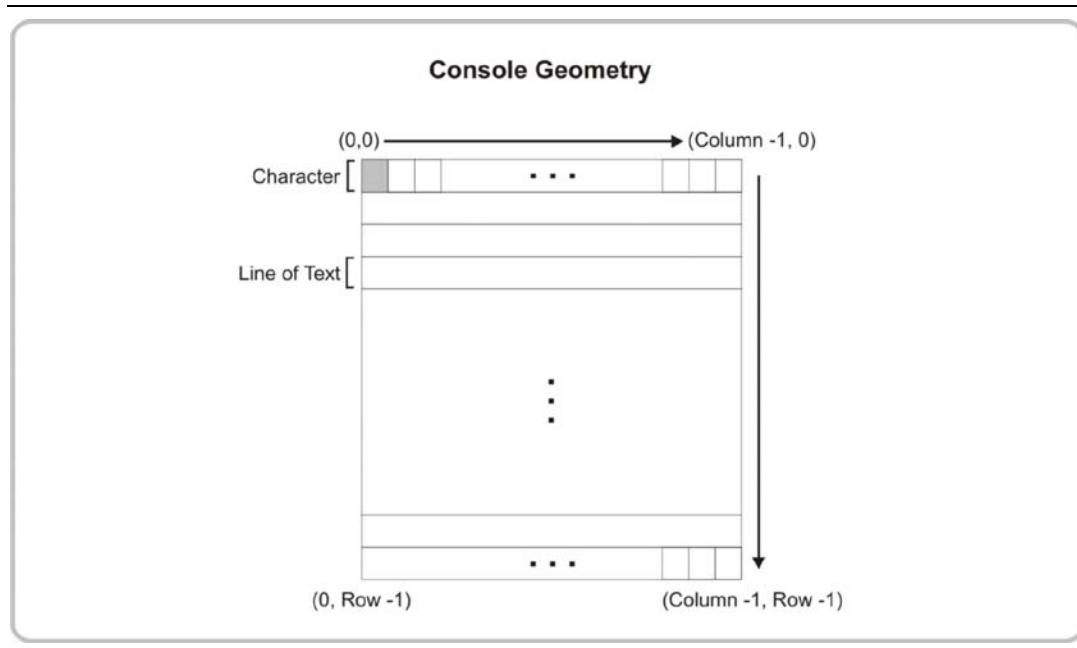


Figure 25—Text Console geometry

22.3.9 `EnableCursor()` Implementation

The `EnableCursor()` function tells the driver whether or not to show the cursor on the console. This has no impact on the position or functionality of the cursor, but only its visible state.

22.4 Serial I/O Protocol Implementations

The implementation of the Serial I/O Protocol is typically found in the file `serialIo.c`. [Appendix A](#) contains a template for a `SerialIo.c` file for a UEFI Driver. The list of tasks to implement the Serial I/O Protocol is as follows:

- Add global variable for the `EFI_SERIAL_IO_PROTOCOL` instance to `SerialIo.c`.
- Add global variable for the `EFI_SERIAL_IO_MODE` structure to `SerialIo.c`.
- Implement the Serial I/O Protocol services in `SerialIo.c`.
- Create a child handle with the Serial I/O Protocol and a Device Path Protocol. The Device Path Protocol chapter of the *UEFI Specification* defines a UART Device Path Node that must be used in the Device Path Protocol for any device that supports the Serial I/O Protocol.

This example shows the protocol interface structure for the Serial I/O Protocol for reference. This protocol is composed of six services, a `Revision` value, and pointer to a `Mode` structure.

```
typedef struct _EFI_SERIAL_IO_PROTOCOL EFI_SERIAL_IO_PROTOCOL;

///
/// The Serial I/O protocol is used to communicate with UART-style serial devices.
/// These can be standard UART serial ports in PC-AT systems, serial ports attached
/// to a USB interface, or potentially any character-based I/O device.
///
struct _EFI_SERIAL_IO_PROTOCOL {
    ///
    /// The revision to which the EFI_SERIAL_IO_PROTOCOL adheres. All future
    /// revisions must be backwards compatible. If a future version is not backwards
    /// compatible, it is not the same GUID.
    ///
    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;
    ///
    /// Pointer to SERIAL_IO_MODE data.
    ///
    EFI_SERIAL_IO_MODE *Mode;
};
```

Example 230—Simple Text Output Protocol

Note: `Mode` must be updated each time that `SetControl()` or `SetAttributes()` is called. This allows the consumers of the Serial I/O Protocol to retrieve the current state of the Serial I/O device.

22.4.1 Reset() Implementation

When this function is called the UEFI Driver must reset the hardware. There is no basic or extended functionality required for this reset function unlike the other reset functions in the console protocols.

22.4.2 SetAttributes() Implementation

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 passes 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 returns `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 must be updated in this function when success has been determined, but not modified if there is an error.

If any attribute is modified that changes any field of the UART Device Path Node for this device, then the Device Path Protocol must be reinstalled with the UEFI Boot Service `ReinstallProtocolInterface()`.

22.4.3 SetControl() and GetControl() Implementation

`GetControl()` and `SetControl()` are used to view and modify respectively the control bits on the serial device. All of the values listed in the following table can be read back with `GetControl()`, but some cannot be modified with `SetControl()`. If a non-modifiable bit is attempted to be set with `SetControl()` then `EFI_UNSUPPORTED` must be returned.

The `Mode` pointer should be updated in this function when success has been determined, but not modified if there is an error.

Table 35—Serial I/O protocol control bits

Control Bit #define	Modifiable with SetControl()
<code>EFI_SERIAL_CLEAR_TO_SEND</code>	NO
<code>EFI_SERIAL_DATA_SET_READY</code>	NO
<code>EFI_SERIAL_RING_INDICATE</code>	NO
<code>EFI_SERIAL_CARRIER_DETECT</code>	NO
<code>EFI_SERIAL_REQUEST_TO_SEND</code>	YES
<code>EFI_SERIAL_DATA_TERMINAL_READY</code>	YES
<code>EFI_SERIAL_INPUT_BUFFER_EMPTY</code>	NO
<code>EFI_SERIAL_OUTPUT_BUFFER_EMPTY</code>	NO
<code>EFI_SERIAL_HARDWARE_LOOPBACK_ENABLE</code>	YES
<code>EFI_SERIAL_SOFTWARE_LOOPBACK_ENABLE</code>	YES
<code>EFI_SERIAL_HARDWARE_FLOW_CONTROL_ENABLE</code>	YES

22.4.4 Write() and Read() Implementation

The `Write()` and `Read()` functions are used to write bytes out to the serial device or read in from the serial device. The only two 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` structure (as set by `SetAttributes()`).

Some serial devices support FIFOs. At the time the `Write()` service is called, the FIFO could be full which means the entire FIFO may need to flush before any new characters can be added to the FIFO. In this case, the time that a UEFI Driver may be required to wait may be longer than the time specified by the `TimeOut` value in the `Mode` structure. The caller is not aware of the FIFO depth, so it is not correct to return an `EFI_TIMEOUT` error if the timeout is due to a full FIFO. Instead, the UEFI Driver should detect the FIFO depth if possible and wait to that number of character times.

Graphics Driver Design Guidelines

This chapter covers the general guidelines for implementing UEFI Drivers for graphics controllers. Most graphics controllers are PCI controllers, and this implies that UEFI Drivers for graphics controllers are typically PCI drivers. PCI drivers must follow all of the PCI design guidelines described in [Chapter 18](#), as well as the general guidelines described in [Chapter 4](#) of this guide. Also see the Rules for PCI/AGP Devices section of the *UEFI Specification*.

If a device is intended to be used as a graphics console output device while UEFI firmware is active, then a UEFI Driver must be implemented that produces the Graphics Output Protocol. The graphics controller must either directly support or be able to emulate the following operations:

- Block transfer to fill a region of the frame buffer
- Block transfer from system memory to region of frame buffer
- Block transfer from region of frame buffer to system memory
- Block transfer between two regions of the frame buffer
- Query attached display devices for EDID information
- Set the supported graphics modes that is intersection of modes that the graphics controller supports and the display device supports

The EDK II provides a platform agnostic driver in the **MdeModulePkg** in the directory **MdeModulePkg/Universal/Console/GraphicsConsoleDxe** that uses the services of a Graphics Output Protocol and bitmap fonts to produce the Simple Text Output Protocol. This means if a Graphics Output Protocol is produced by a UEFI Driver, then the frame buffer managed by that UEFI Driver can be used as a text console device without having to implement the Simple Text Output Protocol in the UEFI Driver for the graphics controller.

23.1 Assumptions

The rest of this chapter assumes that the [Driver Checklist](#) in [Chapter 2](#) has been followed and that the following items have already been identified:

- UEFI Driver Type
- Optional UEFI Driver features
- Supported CPU architectures
- Consumed protocols that are used to produce the graphics controller related protocols.

If the UEFI Driver is required to be compiled for EBC, then see [Chapter 18](#) for PCI optimizations and [Chapter 29](#) for EBC considerations. UEFI Drivers for graphics

controllers are typically more sensitive to the EBC virtual machine interpreter overheads, so it is critical that the performance guidelines are followed for a UEFI Driver for a graphics controller that is compiled for EBC to have good performance.

UEFI Drivers for graphics controllers typically follow the UEFI driver model. Some graphics controllers have a single output controller, and other may have multiple output controllers. In both cases, a child handle must be created for each output controller, which means UEFI Drivers for graphics controllers are always either Bus Drivers or Hybrid Drivers. They are never Device Drivers. UEFI Drivers for graphics controllers are chip-specific because of the requirement to initialize and manage the graphics device.

UEFI drivers that manage graphics controllers typically follow the UEFI Driver Model because the devices are typically on industry standard busses such as PCI. However, it is possible to implement UEFI drivers for graphics controllers that are not on industry standard busses. In these cases, a Root Bridge Driver implementation that produces a handle for each output controller in the driver entry point may be more appropriate than a UEFI Driver Model implementation.

23.2 Graphics Output Protocol Implementation

The implementation of the Graphics Output Protocol is typically found in the file `GraphicsOutput.c`. [Appendix A](#) contains a template for a `GraphicsOutput.c` file for a UEFI Driver. The list of tasks to implement the Graphics Output Protocol is as follows:

- Add global variable for the `EFI_GRAPHICS_OUTPUT_PROTOCOL` instance to `GraphicsOutput.c`.
- Add global variable for the `EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE` structure to `GraphicsOutput.c`.
- Implement the `QueryMode()`, `SetMode()`, and `Blt()` services in `GraphicsOutput.c`.
- Create a child handle for each output display controller and install the Graphics Output Protocol and a Device Path Protocol as described in the Rules for PCI/AGP Devices section of the *UEFI Specification*.
- If a graphics controller has the ability to read EDID information from display devices attached to an output controller, then install the EDID Discovered Protocol with the EDID data on the child handle associated with the output controller.
- Install the EDID Active Protocol with the EDID data on the child handle associated with the output controller. The EDID data comes from either the EDID Override Protocol provided by the platform or the EDID Discovered Protocol.

This example shows the protocol interface structure for the Graphics Output Protocol for reference.

```
typedef struct _EFI_GRAPHICS_OUTPUT_PROTOCOL EFI_GRAPHICS_OUTPUT_PROTOCOL;  
  
///  
/// Provides a basic abstraction to set video modes and copy pixels to and from  
/// the graphics controller's frame buffer. The linear address of the hardware  
/// frame buffer is also exposed so software can write directly to the video  
hardware.
```

```

///+
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;
    ///
    /// Pointer to EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE data.
    ///
    EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE *Mode;
};

extern EFI_GUID gEfiGraphicsOutputProtocolGuid;

```

Example 231—Graphics Output Protocol

23.2.1 Single output graphics adapters

Graphics controllers that are connected to a single output device are the simplest type of UEFI graphics driver. They produce a single child handle and attach both Device Path and Graphics Output protocols onto that handle. They need a single data structure to manage the device. An example of a single output graphics driver stack is shown below.

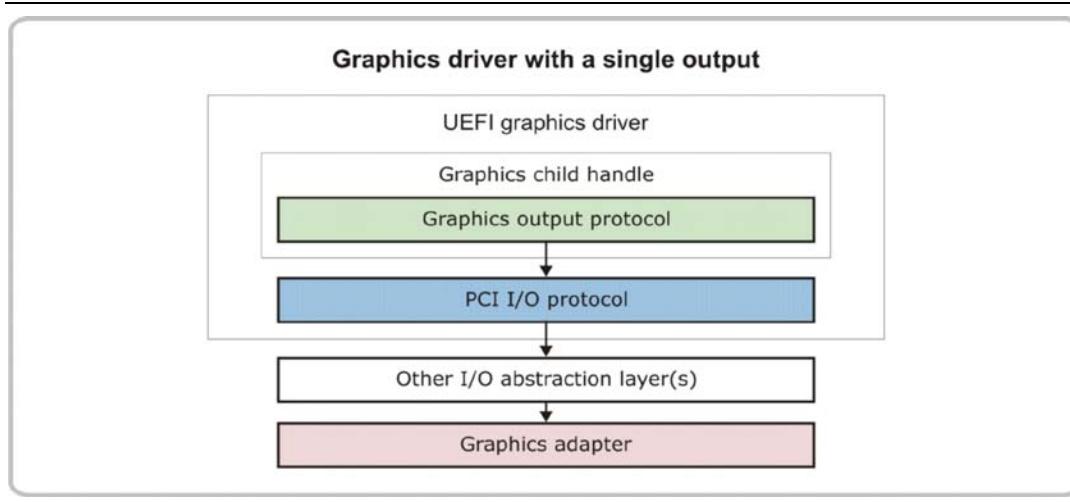


Figure 26—Example single-output graphics driver Implementation

23.2.2 Multiple output graphics adapters

Multiple output graphics drivers (dual or more) are not significantly more complicated than a single channel adapter in UEFI. 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. An example of a dual output graphics adapter [follows](#).

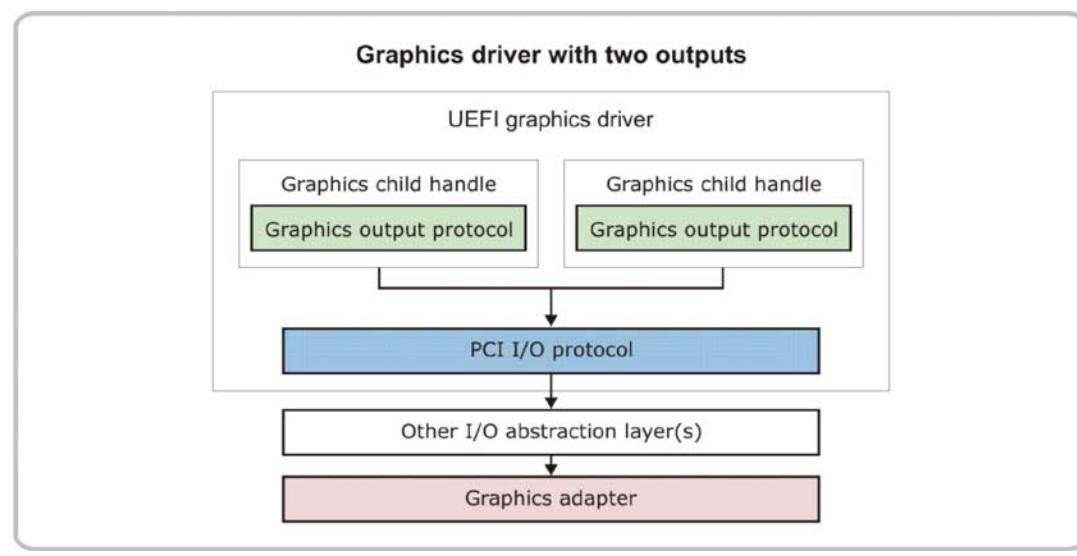


Figure 27—Example dual-output graphics driver implementation

23.2.3 Driver Binding Protocol Implementation

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. The driver must implement all three of the services in the Driver Binding Protocol—`Supported()`, `Start()`, and `Stop()`.

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 conformant 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 the EFI_COMPONENT_NAME_PROTOCOL and EFI_COMPONENT_NAME2_PROTOCOL.
13. In order to support faster boot times, a default mode set and clear screen operation must not be performed in the Start() function. This allows the UEFI Boot Manager to select the best mode for the current boot scenario and set the mode one time.

The `start()` function should not scan for devices every time the driver is started. It should depend on `RemainingDevicePath` 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 `RemainingDevicePath`.

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 RemainingDevicePath, the driver should do so to support the rapid boot capability in the UEFI driver model.

23.2.4 QueryMode(), SetMode(), and Blt() Implementation

There are three functions that make up one method: `QueryMode()`, `SetMode()`, and `Blt()`. 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 has no effect on the hardware or the currently displayed image. It is critical that `QueryMode()` only return modes that can actually be displayed on the attached display device. This means that the UEFI Driver must evaluate the modes that the graphics controller supports and the modes that the attached display supports and only reports the intersection of those two sets. Otherwise, a consumer of the Graphics Output Protocol may attempt to set a mode that cannot be displayed.

The `SetMode()` function is how the consumer of the Graphics Output Protocol selected the specific mode to become active. `SetMode()` is also required to clear the entire display output and reset it all to black.

The `Blt()` function is for transferring information (Block Transfer) to and from the video frame buffer. This is how graphics content is moved to and from the video frame buffer and also allows graphics content to be moved from one location of the video frame buffer to another location of the video frame buffer. The prototype of the `Blt()` function is shown below.

```
/***
 * Blt a rectangle of pixels on the graphics screen. Blt stands for Block Transfer.
 *
 * @param This          Protocol instance pointer.
 * @param BltBuffer     Buffer containing data to blit into video buffer. This
 *                      buffer has a size of Width * Height *
 *                      sizeof(EFI_GRAPHICS_OUTPUT_BLT_PIXEL)
 * @param BltOperation  Operation to perform on BltBuffer and video memory
 * @param SourceX       X coordinate of source for the BltBuffer.
 * @param SourceY       Y coordinate of source for the BltBuffer.
 * @param DestinationX X coordinate of destination for the BltBuffer.
 * @param DestinationY Y coordinate of destination for the BltBuffer.
 * @param Width         Width of rectangle in BltBuffer in pixels.
 * @param Height        Height of rectangle in BltBuffer in pixels.
 * @param Delta         OPTIONAL
 *
 * @retval EFI_SUCCESS           The Blt operation completed.
 * @retval EFI_INVALID_PARAMETER BltOperation is not valid.
 * @retval EFI_DEVICE_ERROR      A hardware error occurred writing to the video
 *                             buffer.
 */
typedef
EFI_STATUS
(EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT)(
    IN  EFI_GRAPHICS_OUTPUT_PROTOCOL          *This,
    IN  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
);
```

Example 232—Graphics Output Protocol Blt() Service

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 be done by performing PCI memory mapped I/O 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.

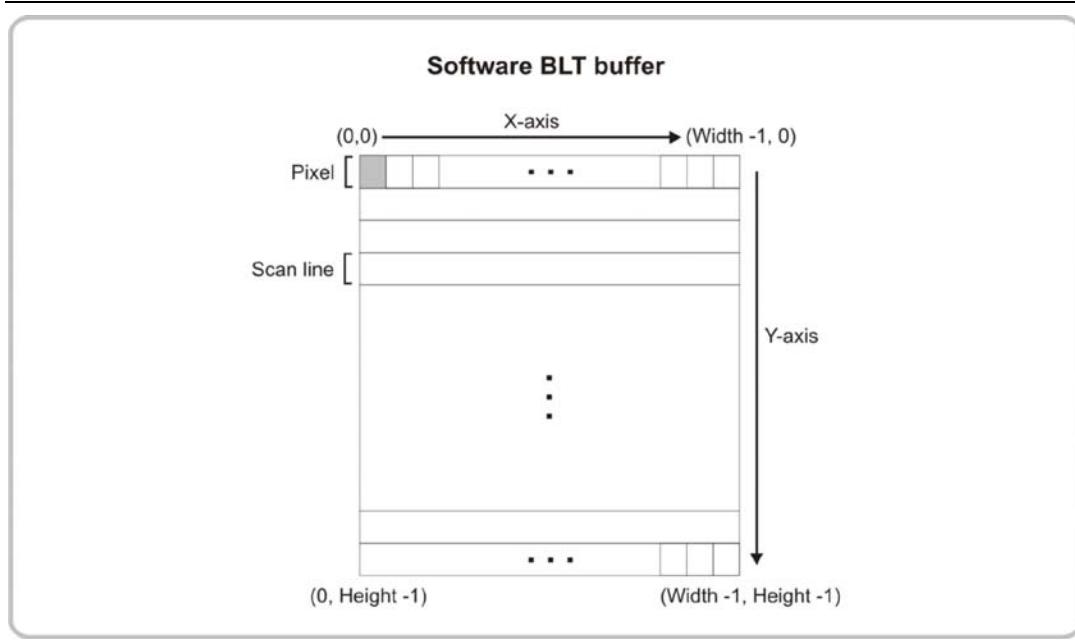


Figure 28—Blt buffer

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 based consoles. A method to scroll the screen can be viewed in EDK II in the GraphicsConsole driver (`\MdeModulePkg\Universal\Console\GraphicsConsoleDxe`).

The `EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE` object pointed to by the `Mode` pointer is populated when the graphics controller is initialized, and must be updated whenever `SetMode()` is called. The `FrameBufferBase` member of this object may be used by a UEFI OS Loader or OS Kernel to update the contents of the graphical display after `ExitBootServices()` is called and the Graphics Output Protocol services are not longer available. A UEFI OS may choose to use this method until an OS driver for the graphics controller can be installed and started.

23.3 EDID Discovered Protocol Implementation

This protocol contains the EDID information that is retrieved from the display device attached to a video output controller. This information may differ from the EDID Active Protocol since the EDID Active Protocol takes into account any interaction with the EDID Override Protocol that was consumed by this driver. This protocol is installed on the child handle that represents a video output and must only represent a single video output device. This protocol does not provide any services. It only provides a pointer to a buffer with the EDID formatted data.

```
///
/// This protocol contains the EDID information retrieved from a video output
/// device.
///
typedef struct {
    ///
    /// The size, in bytes, of the Edid buffer. 0 if no EDID information
    /// is available from the video output device. Otherwise, it must be a
    /// minimum of 128 bytes.
    ///
    UINT32    SizeOfEdid;

    ///
    /// A pointer to a read-only array of bytes that contains the EDID
    /// information for an active video output device. This pointer is
    /// NULL if no EDID information is available for the video output
    /// device. The minimum size of a valid Edid buffer is 128 bytes.
    /// EDID information is defined in the E-DID EEPROM
    /// specification published by VESA (www.vesa.org).
    ///
    UINT8    *Edid;
} EFI_EDID_DISCOVERED_PROTOCOL;

extern EFI_GUID gEfiEdidDiscoveredProtocolGuid;
```

Example 233—EDID Discovered Protocol

23.4 EDID Active Protocol Implementation

The `EFI_EDID_ACTIVE_PROTOCOL` provides information to the system about a video output device. This is 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. The minimum valid size of EDID information is 128 bytes. See the EDID EEPROM specification for details on the format of an EDID. This protocol does not provide any services. It only provides a pointer to a buffer with the EDID formatted data.

```
///
/// This protocol contains the EDID information for an active video output device.
/// This is either the EDID information retrieved from the
/// EFI_EDID_OVERRIDE_PROTOCOL if an override is available, or an identical copy of
/// the EDID information from the EFI_EDID_DISCOVERED_PROTOCOL if no overrides are
/// available.
///
```

```

typedef struct {
    /**
     * The size, in bytes, of the Edid buffer. 0 if no EDID information
     * is available from the video output device. Otherwise, it must be a
     * minimum of 128 bytes.
    /**
    UINT32     SizeOfEdid;

    /**
     * A pointer to a read-only array of bytes that contains the EDID
     * information for an active video output device. This pointer is
     * NULL if no EDID information is available for the video output
     * device. The minimum size of a valid Edid buffer is 128 bytes.
     * EDID information is defined in the E-DID EEPROM
     * specification published by VESA (www.vesa.org).
    /**
    UINT8     *Edid;
} EFI_EDID_ACTIVE_PROTOCOL;

extern EFI_GUID gEfiEdidActiveProtocolGuid;

```

Example 234—EDID Active Protocol

23.5

EDID Override Protocol Implementation

The UEFI platform firmware may produce **EFI_EDID_OVERRIDE_PROTOCOL** in a platform specific driver implementation. This implementation of this protocol is not the responsibility of the UEFI Driver that produces the Graphics Output Protocol. If the UEFI platform firmware produces this protocol, then UEFI Driver for a graphics controller must use this information when producing the EDID Active Protocol on the same handle as the Graphics Output Protocol.

The implementation of the EDID Override Protocol is typically found in the file **EdidOverride.c**. [Appendix A](#) contains a template for an **EdidOverride.c** file for a platform specific UEFI Driver. The list of tasks to implement the EDID Override Protocol is as follows:

- Add global variable for the **EFI_EDID_OVERRIDE_PROTOCOL** instance to **EdidOverride.c**.
- Implement the **GetEdid()** service in **EdidOverride.c**.
- The implementation of the EDID Override Protocol is typically in a Service Driver. This means that the EDID Override Protocol is typically installed onto a new handle in the Handle Database in the platform specific driver's entry point.

The following example shows the protocol interface structure for the EDID Override Protocol for reference.

```

typedef struct _EFI_EDID_OVERRIDE_PROTOCOL EFI_EDID_OVERRIDE_PROTOCOL;

< /**
   * This protocol is produced by the platform to allow the platform to provide
   * EDID information to the producer of the Graphics Output protocol.
  /**
struct _EFI_EDID_OVERRIDE_PROTOCOL {
    EFI_EDID_OVERRIDE_PROTOCOL_GET_EDID  GetEdid;
};

extern EFI_GUID gEfiEdidOverrideProtocolGuid;

```

Example 235—DID Override Protocol

23.5.1 GetEdid() Implementation

The `GetEdid()` function returns the handle, attributes (override always, never, only if nothing is returned), and the new EDID information. This is then used by UEFI Drivers for graphics controller to produce the EDID Active Protocol.

Mass Storage Driver Design Guidelines

This chapter covers the general guidelines for implementing UEFI Drivers for mass storage devices. Most mass storage devices reside on industry standard busses such as ATA, SCSI, or USB. This means that the design guidelines as described in [Chapter 21](#) for ATA, [Chapter 20](#) for SCSI, or [Chapter 19](#) for USB must be followed along with the general guidelines described in [Chapter 4](#) of this guide.

If a mass storage device is intended to be used as a boot device for a UEFI operating system or UEFI applications, then a UEFI Driver must be implemented that produces the Block I/O Protocol. If the UEFI Driver is required to be conformant with the *UEFI Specification* 2.3.1 or higher, then the Block I/O 2 Protocol must also be produced. If the mass storage device supports the SPC-4 or ATA8-ACS security commands, then the Storage Security Command Protocol must also be produced. A mass storage device must either directly support or be able to emulate the following operations:

- Read blocks of data from the mass storage device.
- Write blocks of data to the mass storage device.
- Determine the size of the blocks on the mass storage device.
- Determine the total number of blocks on the mass storage device.
- If the mass storage device supports removable media, then methods must exist to determine if media is present, media is not present, and if the media has been changed.

If a mass storage device does not meet these requirements, but still must support being used as a boot device, then consider implementing a UEFI Driver that produces either the Simple File System Protocol or the Load File Protocol. Please see the Media Access chapter of the *UEFI Specification* for details on the Simple File System Protocol and [Chapter 27](#) for details on the Load File Protocol.

The EDK II provides a set of platform agnostic drivers in the [MdeModulePkg](#) and the [FatBinPkg](#) that consume the Block I/O Protocols and produce the Simple File System Protocol which is one of the two protocols from which a UEFI Boot Manager is able to boot a UEFI operating system or a UEFI application. These platform agnostic drivers allow the contents of the mass storage media to be accessed without any specialized knowledge of the specific device or controller. The set platform agnostic drivers UEFI Drivers include:

- [MdeModulePkg/Universal/Disk/DiskIoDxe](#)
- [MdeModulePkg/Universal/Disk/PartitionDxe](#)
- [MdeModulePkg/Universal/Disk/UnicodeCollation](#)
- [FatBinPkg/EnhancedFatDxe](#)

24.1 Assumptions

The rest of this chapter assumes that the Driver Checklist in [Chapter 2](#) has been followed and that the following items have already been identified:

- UEFI Driver Type
- Optional UEFI Driver features
- Supported CPU architectures
- Consumed protocols that are used to produce the mass storage device related protocols.

UEFI drivers that manage mass storage devices typically follow the UEFI Driver Model because these devices are typically on industry standard busses such as USB, SCSI, or ATA. However, it is possible to implement UEFI Drivers for mass storage devices that are not on industry standard busses supported by the *UEFI Specification*. In these cases, a Root Bridge Driver implementation that produces a handles for mass storage devices in the driver entry point may be more appropriate than a UEFI Driver Model implementation.

24.2 Block I/O Protocol Implementations

The implementation of the Block I/O Protocols is typically found in the file `Block.c`. [Appendix A](#) contains a template for a `BlockIo.c` file for a UEFI Driver. The list of tasks to implement the Block I/O Protocols is as follows:

- Add global variable for the `EFI_BLOCK_IO_PROTOCOL` instance to `BlockIo.c`.
- Add global variable for the `EFI_BLOCK_IO2_PROTOCOL` instance to `BlockIo.c`.
- Add global variable for the `EFI_BLOCK_IO_MODE` structure to `BlockIo.c`.
- Implement the Block I/O Protocol and Block I/O 2 Protocol services in `BlockIo.c`.

Example 236, below, shows the protocol interface structure for the Block I/O Protocol and the following Example 237 shows the protocol interface structure for the Block I/O 2 Protocol for reference. These two protocols are very similar and are both composed of four services and a pointer to a structure that provides detailed information on the currently mounted media. The main difference between these two protocols is that the Block I/O 2 Protocol supports non-blocking operations.

```
typedef struct _EFI_BLOCK_IO_PROTOCOL  EFI_BLOCK_IO_PROTOCOL;
/// This protocol provides control over block devices.
struct _EFI_BLOCK_IO_PROTOCOL {
    /// The revision to which the block IO interface adheres. All future
    /// revisions must be backwards compatible. If a future version is not
    /// back wards compatible, it is not the same GUID.
    ///
    UINT64          Revision;
    ///
    /// Pointer to the EFI_BLOCK_IO_MEDIA data for this device.
}
```

```

/// 
EFI_BLOCK_IO_MEDIA *Media;

EFI_BLOCK_RESET      Reset;
EFI_BLOCK_READ       ReadBlocks;
EFI_BLOCK_WRITE      WriteBlocks;
EFI_BLOCK_FLUSH      FlushBlocks;

};

extern EFI_GUID gEfiBlockIoProtocolGuid;

```

Example 236—Block I/O Protocol

Note: `Media` must be updated each time that that media in the mass storage device is inserted or removed. This allows the consumers of the Block I/O Protocol to retrieve the state of the currently mounted media.

```

typedef struct _EFI_BLOCK_IO2_PROTOCOL  EFI_BLOCK_IO2_PROTOCOL;

///
/// The Block I/O2 protocol defines an extension to the Block I/O protocol which
/// enables the ability to read and write data at a block level in a non-blocking
// manner.
///
struct _EFI_BLOCK_IO2_PROTOCOL {
    ///
    /// A pointer to the EFI_BLOCK_IO_MEDIA data for this device.
    /// Type EFI_BLOCK_IO_MEDIA is defined in BlockIo.h.
    ///
    EFI_BLOCK_IO_MEDIA *Media;

    EFI_BLOCK_RESET_EX   Reset;
    EFI_BLOCK_READ_EX    ReadBlocksEx;
    EFI_BLOCK_WRITE_EX   WriteBlocksEx;
    EFI_BLOCK_FLUSH_EX   FlushBlocksEx;
};

extern EFI_GUID gEfiBlockIo2ProtocolGuid;

```

Example 237—Block I/O 2 Protocol

Note: `Media` must be updated each time that that media in the mass storage device is inserted or removed. This allows the consumers of the Block I/O 2 Protocol to retrieve the state of the currently mounted media.

24.2.1 Reset() Implementation

The `Reset()` function resets the block device hardware. During this operation the UEFI Driver must ensure 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.

24.2.2 ReadBlocks() and ReadBlocksEx() Implementation

Reading blocks from media typically uses the following order of operations:

1. Verify media presence. This is critical for removable or swappable media.
2. If a media change event is detected, then reinstall the Block I/O Protocols using the UEFI Boot Service `ReinstallProtocolInterface()`. A media change event can be a change from the media present state to the media not present state. A change from the media not present state to the media present state. The `BlockSize` field of the `Media` structure must have a non-zero value, even when no media is present.
 - If there is no media, return `EFI_NO_MEDIA`.
 - If the media is different, return `EFI_MEDIA_CHANGED`.
3. Verify parameters
 - The `Buffer`, sized `BufferSize`, must be 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 requested sectors from the media
 - If a non-blocking request is made through `ReadBlocksEx()`, then start the request and if the request is expected to take some time to complete, set up a timer event to periodically check the completion status and return immediately. When the request is complete, signal the event passed into `ReadBlocksEx()` to inform the caller that the previous request has been completed.
5. If needed, copy the appropriate portion of the read into `Buffer`.
 - (Optional) Update the driver's cache for better performance.

24.2.3 WriteBlocks() and WriteBlockEx() Implementation

Writing blocks from media typically uses the following order of operations:

1. Verify media presence. This is critical for removable or swappable media.
2. If a media change event is detected, then reinstall the Block I/O Protocols using the UEFI Boot Service `ReinstallProtocolInterface()`. A media change event can be a change from the media present state to the media not present state. A change from the media not present state to the media present state. A change from the media present state to the media present state with different media in the device being managed.
3. If there is no media return `EFI_NO_MEDIA`.

4. If the media is different return EFI_MEDIA_CHANGED
5. Get the media's block size. The BlockSize field of the Media structure must have a non-zero value, even when no media is present.
6. Verify parameters.
7. The Buffer, sized BufferSize, is a whole number of blocks.
8. The write does not start past the end of the media.
9. The write does not extend past the end of the media.
10. The Buffer is aligned as required.
11. If needed, copy the appropriate portion of the buffer to a location visible to the mass storage device.
12. Write the appropriate sectors to the media
13. If a non-blocking request is made through WriteBlocksEx(), then start the request and if the request is expected to take some time to complete, set up a timer event to periodically check the completion status and return immediately. When the request is complete, signal the event passed into WriteBlocksEx() to inform the caller that the previous request has been completed.
14. (Optional) Update the driver's cache for better performance.

24.2.4 **FlushBlocks() and FlushBlocksEx() Implementation**

FlushBlocks() and **FlushBlocksEx()** are used to ensure that all pending writes have been completed on the mass storage device. This can be used as part of a check before removing some media from the system. Combinations of both read and write operations may be performed as part of this operation.

If a non-blocking request is made through **FlushBlocksEx()**, then start the request and if the request is expected to take some time to complete, set up a timer event to periodically check the completion status and return immediately. When the request is complete, signal the event passed into **FlushBlocksEx()** to inform the caller that the previous request has been completed.

24.3 **Storage Security Protocol Implementation**

The implementation of the Storage Security Protocol is only required if the mass storage device supports the SPC-4 or ATA8-ACS security commands. The implementation of the Storage Security Protocol is typically found in the file **Block.c**. [Appendix A](#) contains a template for a **BlockIo.c** file for a UEFI Driver. The list of tasks to implement the Storage Security Protocol is as follows:

- Add global variable for the `EFI_STORAGE_SECURITY_COMMAND_PROTOCOL` instance to `BlockIo.c`.
- Implement the Storage Security Command Protocol services in `BlockIo.c`.

This example shows the protocol interface structure for the optional Storage Security Command Protocol for reference. It is composed of two services to send and receive data.

```
typedef struct _EFI_STORAGE_SECURITY_COMMAND_PROTOCOL
  EFI_STORAGE_SECURITY_COMMAND_PROTOCOL;

///
/// The EFI_STORAGE_SECURITY_COMMAND_PROTOCOL is used to send security protocol
/// commands to a mass storage device. Two types of security protocol commands
/// are supported. SendData sends a command with data to a device. ReceiveData
/// sends a command that receives data and/or the result of one or more commands
/// sent by SendData.
///
/// The security protocol command formats supported shall be based on the
/// definition of the SECURITY PROTOCOL IN and SECURITY PROTOCOL OUT commands
/// defined in SPC-4. If the device uses the SCSI command set, no translation is
/// needed in the firmware and the firmware can package the parameters into a
/// SECURITY PROTOCOL IN or SECURITY PROTOCOL OUT command and send the command to
/// the device. If the device uses a non-SCSI command set, the firmware shall map
/// the command and data payload to the corresponding command and payload format
/// defined in the non-SCSI command set (for example, TRUSTED RECEIVE and TRUSTED
/// SEND in ATA8-ACS).
///
/// The firmware shall automatically add an EFI_STORAGE_SECURITY_COMMAND_PROTOCOL
/// for any storage devices detected during system boot that support SPC-4,
/// ATA8-ACS or their successors.
///
struct _EFI_STORAGE_SECURITY_COMMAND_PROTOCOL {
  EFI_STORAGE_SECURITY_RECEIVE_DATA  ReceiveData;
  EFI_STORAGE_SECURITY_SEND_DATA     SendData;
};

extern EFI_GUID gEfiStorageSecurityCommandProtocolGuid;
```

Example 238—Storage Security Command Protocol

The EDK II has a complete implementation of the Storage Security Protocol for ATA device in the `MdeModulePkg` in the directory `MdeModulePkg/Bus/Ata/AtaBusDxe`. This can be used as a reference for implementations of the Storage Security Protocol for mass storage devices on other bus types.

Network Driver Design Guidelines

This chapter focuses on the design and implementation of UEFI Drivers for network interface controllers. These UEFI Drivers typically bus drivers follow the UEFI Driver Model. Some example devices include add-in PCI network adapters, USB network controllers, cardbus network cards, and LAN-on-motherboard network devices. This list illustrates that most network interface controllers are PCI devices or USB devices. As a result, the UEFI Drivers for network interface controllers must follow all of the design guidelines described in [Chapter 18](#) for PCI or [Chapter 19](#) for USB, and must also follow the general guidelines described in [Chapter 4](#).

If a network interface controller is intended to be used as a boot device for a UEFI operating system or UEFI applications, then a UEFI Driver must be implemented that produces Network Interface Identifier Protocol and UNDI, the Simple Network Protocol, or the Managed Network Protocol. If the network interface controller hardware supports VLAN, then the VLAN Config Protocol must be implemented. If the UEFI Driver for a network interface controller only produces the Managed Network Protocol, then the UEFI Driver must also produce the VLAN Config Protocol even if the network interface controller does not support VLAN.

All three UEFI Driver designs for network interface controllers are covered in this chapter. There are several factors that affect the design of a UEFI Driver for a network interface controller. The following table summarizes the major features for each of the three possible UEFI Driver designs.

Table 36—Network driver differences

Feature	II and UNDI	imple Network Protocol	anaged Network Protocol
UEFI Runtime Driver	Yes	No	No
Depends on platform agnostic UEFI Driver for Simple Network Protocol	Yes	No	No
Depends on platform agnostic UEFI Driver for Managed Network Protocol	Yes	Yes	No
Requires UNDI interface	Yes	No	No
Supports EBC CPU Architecture	No	Yes	Yes
Requires Exit Boot Services Event	Yes	Maybe	Maybe
Requires Set Virtual Address Map Event	Yes	No	No

The EDK II provides a set of platform agnostic drivers in the [MdeModulePkg](#) and the [NetworkPkg](#) that consume the protocols produced by a UEFI Driver for a network interface controller and produce the Load File Protocol which is one of the two protocols

from which a UEFI Boot Manager is able to boot a UEFI operating system or a UEFI application. The Load File Protocol allows a UEFI operating system or UEFI application to be booted from a properly configured PXE server. The platform agnostic drivers allow the services provided by the network interface controller to be accessed without any specialized knowledge of the specific device or controller. The set platform agnostic UEFI Drivers include:

- **MdeModulePkg/Universal/Network/ArpDxe**
- **MdeModulePkg/Universal/Network/Dhcp4Dxe**
- **MdeModulePkg/Universal/Network/DpcDxe**
- **MdeModulePkg/Universal/Network/Ip4ConfigDxe**
- **MdeModulePkg/Universal/Network/Ip4Dxe**
- **MdeModulePkg/Universal/Network/IScsiDxe**
- **MdeModulePkg/Universal/Network/MnpDxe**
- **MdeModulePkg/Universal/Network/Mtftp4Dxe**
- **MdeModulePkg/Universal/Network/SnpDxe**
- **MdeModulePkg/Universal/Network/Tcp4Dxe**
- **MdeModulePkg/Universal/Network/Udp4Dxe**
- **MdeModulePkg/Universal/Network/UefiPxeBcDxe**
- **MdeModulePkg/Universal/Network/VlanConfigDxe**
- **NetworkPkg/Dhcp6Dxe**
- **NetworkPkg/Ip6Dxe**
- **NetworkPkg/IpSecDxe**
- **NetworkPkg/IScsiDxe**
- **NetworkPkg/Mtftp6Dxe**
- **NetworkPkg/TcpDxe**
- **NetworkPkg/Udp6Dxe**
- **NetworkPkg/UefiPxeBcDxe**

These platform agnostic drivers also provide support for iSCSI which produces the Block I/O Protocol for a network boot target. Additional platform agnostic drivers produce the Simple File System Protocol from a Block I/O Protocol. Those details are provided in [Chapter 24](#) on mass storage devices.

25.4 Assumptions

The rest of this chapter assumes that the [Driver Checklist](#) in Chapter 2 has been followed and that the following items have already been identified:

- UEFI Driver Type
- Optional UEFI Driver features
- Supported CPU architectures
- Consumed protocols that are used to produce the network interface controller related protocols.

UEFI drivers that manage network interface controllers typically follow the UEFI Driver Model because the devices are typically on industry standard busses such as PCI or USB. However, it is possible to implement UEFI drivers for network interface controllers that are not on industry standard busses. In these cases, a Root Bridge Driver implementation that produces a handle for network interface controller in the driver entry point may be more appropriate than a UEFI Driver Model implementation.

25.5 NII Protocol and UNDI Implementations

Network drivers that follow the UNDI definition from the *UEFI Specification* are unique compared to all others peripheral drivers.

- UEFI Drivers that produce UNDI interfaces must be UEFI Runtime Drivers. This allows a UEFI operation system to potentially use the services of this UEFI Runtime Driver to provide basic network connectivity in boot scenarios where the OS driver for the network interface controller is not available.
- UNDI is not a protocol interface. The Network Interface Identifier Protocol defines the entry point to the UNDI structure, but UNDI itself is not a protocol. The Command Descriptor Block (CDB) that the caller passed into each UNDI request must provide services that allow the UNDI to access the network interface controller hardware.
- See the Universal Network Driver Interfaces appendix of the *UEFI Specification* for more details on UNDI adapters.

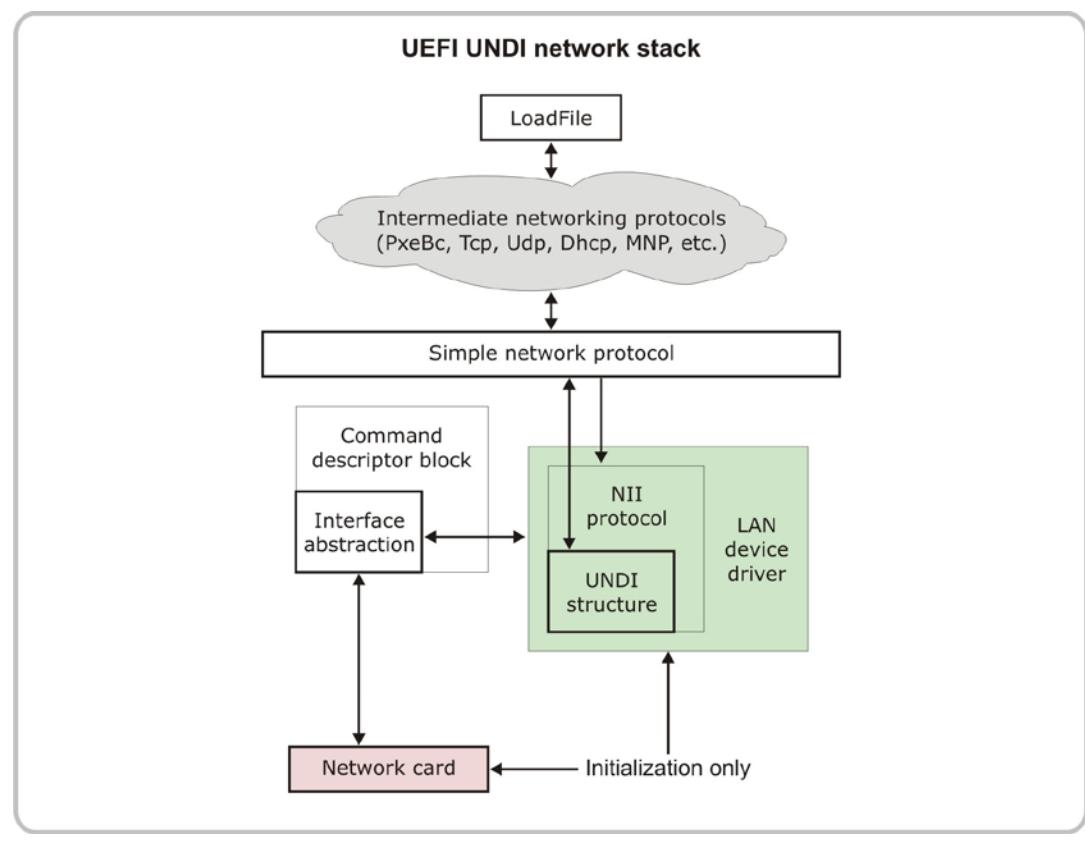


Figure 29—UEFI UNDI Network Stack

The implementation of the Network Interface Identifier Protocol is typically found in the file [NiiUndi.c](#). [Appendix A](#) contains a template for a [NiiUndi.c](#) file for a UEFI Driver.

The list of tasks to implement the Network Interface Identifier Protocol and UNDI is as follows:

- Add global variable for the `EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL` instance to [NiiUndi.c](#).
- Implement the UNDI interface in [NiiUndi.c](#).
- Create child handle in Driver Binding Protocol `start()` and install the NII Protocol and the Device Path Protocol.
- Create an Exit Boot Services Event to disable DMA when packets are received.
- Create a Set Virtual Address Map Event to convert physical addresses to virtual addresses.

The [following example](#) shows the protocol interface structure for the Network Interface Identifier Protocol for reference. The Network Interface Identifier Protocol is different from many other protocols in that it has no functions inside it, and instead is only composed of data fields. These data fields share information with the platform about the network interface controller capabilities. The field called `id` provides the address of a data structure for the UNDI that includes methods for the platform to call the UNDI interfaces.

```

typedef struct _EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL
  EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL;

///
/// An optional protocol that is used to describe details about the software
/// layer that is used to produce the Simple Network Protocol.
///
struct _EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL {
  ///
  /// The revision of the EFI_NETWORK_INTERFACE_IDENTIFIER protocol.
  ///
  UINT64      Revision;
  ///
  /// The address of the first byte of the identifying structure for this network
  /// interface. This is only valid when the network interface is started
  /// (see Start()). When the network interface is not started, this field is set
  /// to zero.
  ///
  UINT64      Id;
  ///
  /// The address of the first byte of the identifying structure for this
  /// network interface. This is set to zero if there is no structure.
  ///
  UINT64      ImageAddr;
  ///
  /// The size of unrelocated network interface image.
  ///
  UINT32      ImageSize;
  ///
  /// A four-character ASCII string that is sent in the class identifier field of
  /// option 60 in DHCP. For a Type of EfiNetworkInterfaceUndi, this field is UNDI.
  ///
  CHAR8       StringId[4];
  ///
  /// Network interface type. This will be set to one of the values
  /// in EFI_NETWORK_INTERFACE_TYPE.
  ///
  UINT8       Type;
  ///
  /// Major version number.
  ///
  UINT8       MajorVer;
  ///
  /// Minor version number.
  ///
  UINT8       MinorVer;
  ///
  /// TRUE if the network interface supports IPv6; otherwise FALSE.
  ///
  BOOLEAN     Ipv6Supported;
  ///
  /// The network interface number that is being identified by this Network
  /// Interface Identifier Protocol. This field must be less than or equal
  /// to the IfCnt field in the !PXE structure.
  ///
  UINT8       IfNum;
};

extern EFI_GUID gEfiNetworkInterfaceIdentifierProtocolGuid_31;

```

Example 239—Network Interface Identifier Protocol

The [following table](#) shows the data structure called **!PXE** that resides at address specified by the **rd** field of the Network Interface Identifier Protocol.

Table 37—!PXE interface 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)					

This table shows the layout of the Command Descriptor Block (CDB) structure that is passed into the function specified by the Entry Point field of the !PXE structure.

Table 38—CDB structure

Command descriptor bock (CDB)					
Offset	0x00	0x01	0x02	0x03	
0x00	OpCode		OpFlags		
0x04	CPBsize		DBsize		
0x08	CPBaddr				
0x0C					
0x10	DBaddr				
0x14					
0x18	StatCode		StatFlags		
0x1C	IFnum		Control		

The UEFI Driver for a network interface controller that implements an UNDI must implement all the UNDI related OpCodes required by the *UEFI Specification*.

25.5.1 Exit Boot Services Event

UEFI Drivers for network interface controllers that perform DMA operations to a buffer in system memory in response to a received packet must create an Exit Boot Services Event in the Driver Binding Protocol `start()` function. The notification function associated with this Exit Boot Services Event must update the network interface controller hardware to disable all further DMA activity. This guarantees that after `ExitBootServices()` is called, that the receive resources allocated to network driver are freed for OS usage.

Caution: If the network interface controller performs DMA due to received packets into system memory after `ExitBootServices()` is called, the DMA operations may corrupt memory that is now owned by the operating system.

25.5.2 Set Virtual Address Map Event

If a UEFI Runtime Driver dynamically allocates memory buffers, then the pointers to those allocations and pointers within those allocations must be converted to virtual addresses when a UEFI operating system calls the UEFI Runtime Service `SetVirtualAddressMap()`. UEFI Drivers for network interface controllers that manage this type of buffer must create a Set Virtual Address Map Event in the Driver Binding Protocol `start()` function. The notification function associated with this Set Virtual Address Map Event must use the UEFI Runtime Service called `ConvertPointer()` to perform conversions from physical addresses to virtual addresses on all pointers. These conversions must be performed bottom-up since the virtual pointers are not valid until the `SetVirtualAddressMap()` returns to the UEFI operating system.

25.5.3 Memory leaks caused by UNDI

UNDI drivers transfer data in the system through memory buffers. To perform its function, the UNDI driver often allocates many buffers for data transfer. If those buffers are not tracked properly, it is possible to lose them in the shuffle, and they are not returned to the system memory management. This can cause a memory leak. When a buffer is being used (taken off the waiting queue and made active) there is a chance of losing the pointer to that buffer in the process, which again, causes a memory leak.

When transmitting, the UNDI driver must keep track of which buffers have been completed, and 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.

25.6 Simple Network Protocol Implementations

Exposing SNP instead of NII and UNDI has some advantages and some disadvantages over using NII and UNDI. SNP-based network drivers are never UEFI Runtime Drivers, so such drivers do not have to worry about meeting the UEFI Runtime Driver requirements. This allows an SNP driver to be compiled for all the CPU architectures supported by the *UEFI Specification* including EBC. SNP may be required for some non-standard network interface controllers that cannot meet the UNDI requirements.

When a network driver exposes SNP directly the system firmware layers MNP on top of SNP and does 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 Load File Protocol is not guaranteed here, but is a choice made by the system firmware.

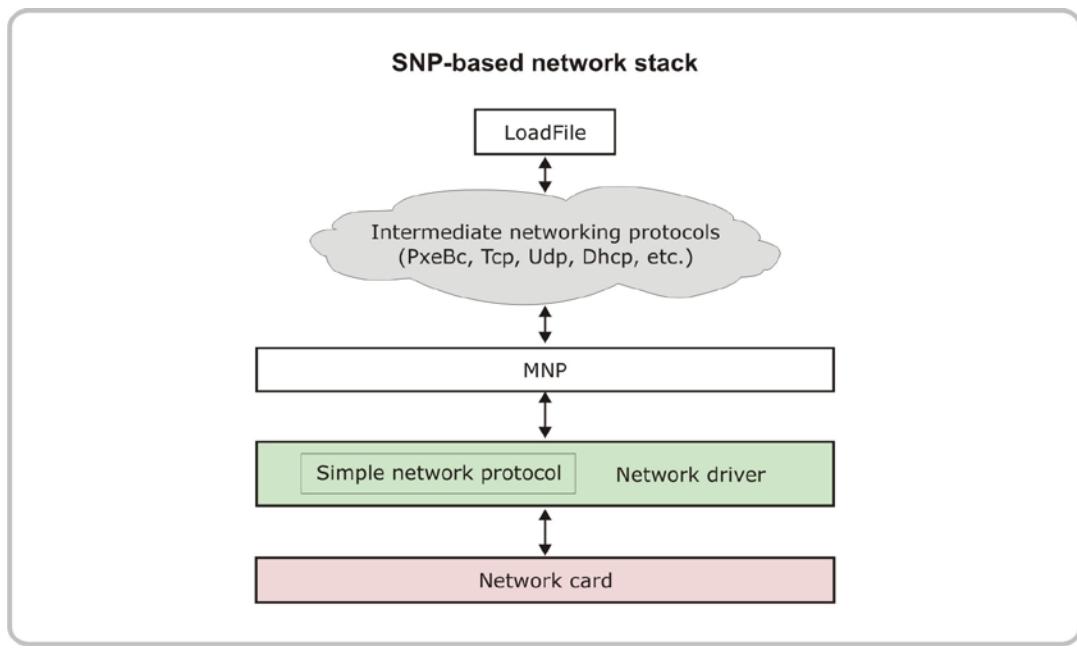


Figure 30—SNP-based network stack

The implementation of the Simple Network Protocol is typically found in the file `SimpleNetwork.c`. [Appendix A](#) contains a template for a `SimpleNetwork.c` file for a UEFI Driver. The list of tasks to implement the Simple Network Protocol is as follows:

- Add global variable for the `EFI_SIMPLE_NETWORK_PROTOCOL` instance to `SimpleNetworkProtocol.c`.
- Create child handle in Driver Binding Protocol `start()` and install the Simple Network Protocol and the Device Path Protocol. Also allocate and initialize an `EFI_SIMPLE_NETWORK_MODE` structure in the Simple Network Protocol.
- Implement the Simple Network Protocol services in `SimpleNetwork.c`.
- Create an Exit Boot Services Event to disable DMA when packets are received.

The following example shows the protocol interface structure for the Simple Network Protocol for reference. This protocol is composed of 13 services, an `EFI_EVENT` that can be used to poll when a packet has been received, and a `Mode` structure that contains details on the attributes and capabilities of the network interface controller.

```

typedef struct _EFI_SIMPLE_NETWORK_PROTOCOL EFI_SIMPLE_NETWORK_PROTOCOL;

///
/// The EFI_SIMPLE_NETWORK_PROTOCOL protocol is used to initialize access
/// to a network adapter. Once the network adapter initializes,
/// the EFI_SIMPLE_NETWORK_PROTOCOL protocol provides services that
/// allow packets to be transmitted and received.
///
struct _EFI_SIMPLE_NETWORK_PROTOCOL {
    ///

```

```

/// Revision of the EFI_SIMPLE_NETWORK_PROTOCOL. All future revisions must
/// be backwards compatible. If a future version is not backwards compatible
/// it is not the same GUID.
///
UINT64                                Revision;
EFI_SIMPLE_NETWORK_START                 Start;
EFI_SIMPLE_NETWORK_STOP                  Stop;
EFI_SIMPLE_NETWORK_INITIALIZE           Initialize;
EFI_SIMPLE_NETWORK_RESET                 Reset;
EFI_SIMPLE_NETWORK_SHUTDOWN              Shutdown;
EFI_SIMPLE_NETWORK_RECEIVE_FILTERS      ReceiveFilters;
EFI_SIMPLE_NETWORK_STATION_ADDRESS       StationAddress;
EFI_SIMPLE_NETWORK_STATISTICS            Statistics;
EFI_SIMPLE_NETWORK_MCAST_IP_TO_MAC      MCastIpToMac;
EFI_SIMPLE_NETWORK_NVDATA                NvData;
EFI_SIMPLE_NETWORK_GET_STATUS             GetStatus;
EFI_SIMPLE_NETWORK_TRANSMIT               Transmit;
EFI_SIMPLE_NETWORK_RECEIVE                Receive;
///
/// Event used with WaitForEvent() to wait for a packet to be received.
///
EFI_EVENT                                WaitForPacket;
///
/// Pointer to the EFI_SIMPLE_NETWORK_MODE data for the device.
///
EFI_SIMPLE_NETWORK_MODE                  *Mode;
};

extern EFI_GUID gEfiSimpleNetworkProtocolGuid;

```

Example 240—Simple Network Protocol

25.7 Managed Network Protocol Implementations

Exposing MNP instead has many of the same advantages of implementing SNP and it reduces one extra layer of drivers. One disadvantage of implementing MNP instead of SNP, or NII and UNDI is that the VLAN Config Protocol must also be implemented. In addition, the Managed Network Protocol also requires a Service Binding Protocol to be implemented. See [Chapter 10](#) covering the Service Binding Protocol. In many cases, since two additional protocols must be implemented in addition to the Managed Network Protocol, it is **recommended** that the Simple Network Protocol be implemented instead of the Managed Network Protocol.

User Credential Driver Design Guidelines

The User Credential Protocol provides a method to identify the user of a platform. If a device provides a method to identify the user of a platform such as entering a password, reading a fingerprint, or reading a smart token, then a UEFI Driver that produces the User Credential Protocol should be implemented.

The EDK II provides the following two implementations of the User Credential Protocol in the `SecurityPkg`. The first one interacts with the user to retrieve a password entered through a keyboard. The second one uses a content stored on a USB Flash drive as a token.

- `SecurityPkg\UserIdentification\PwdCredentialProviderDxe`
- `SecurityPkg\UserIdentification\UsbCredentialProviderDxe`

26.1 Assumptions

The rest of this chapter assumes that the Driver Checklist in [Chapter 2](#) has been followed and that the following items have already been identified:

- UEFI Driver Type
- Optional UEFI Driver features
- Supported CPU architectures
- Consumed protocols that are used to produce the User Credential Protocol.

UEFI drivers that produce the User Credential Protocol typically follow the UEFI Driver Model. However, it is possible to implement UEFI Drivers that directly produce the User Credential Protocol for a single device in a platform or a software only based identification method. In this case a Root Bridge Driver implementation may be more appropriate than a UEFI Driver Model implementation.

26.2 User Credential Protocol Implementation

The implementation of the User Credential Protocol is typically found in the file `UserCredential.c`. [Appendix A](#) contains a template for a `UserCredential.c` file for a UEFI Driver. The list of tasks to implement the User Credential Protocol is as follows:

- Add global variable for the `EFI_USER_CREDENTIAL2_PROTOCOL` instance to `UserCredential.c`.
- Add implementations of the services produced by the User Credential Protocol to `UserCredential.c`.

- Implement HII forms for interacting with the user during the user identify process using a formset GUID of [EFI_USER_CREDENTIAL_PROTOCOL_GUID](#). See [Chapter 12](#) for details on HII forms.
- Implement HII Config Access Protocol to retrieve and save configuration information associated with the HII forms. See [Chapter 12](#) for details on the HII Config Access Protocol. The implementation of the HII Config Access Protocol is typically found in the file [HiiConfigAccess.c](#). [Appendix A](#) contains a template for a [HiiConfigAccess.c](#) file for a UEFI Driver.

The example below shows the protocol interface structure for the User Credential Protocol for reference. This protocol is composed of two GUIDs, 11 services, and a capabilities value. These services are used by a User Identity Manager to identify the current user of a platform.

```
typedef struct _EFI_USER_CREDENTIAL2_PROTOCOL  EFI_USER_CREDENTIAL2_PROTOCOL;
///
/// This protocol provides support for a single class of credentials
///
struct _EFI_USER_CREDENTIAL2_PROTOCOL {
    EFI_GUID           Identifier;   ///< Uniquely identifies this
                                      ///< credential provider.
    EFI_GUID           Type;        ///< Identifies this class of User
                                      ///< Credential Provider.
    EFI_CREDENTIAL2_ENROLL    Enroll;
    EFI_CREDENTIAL2_FORM     Form;
    EFI_CREDENTIAL2_TILE     Tile;
    EFI_CREDENTIAL2_TITLE    Title;
    EFI_CREDENTIAL2_USER     User;
    EFI_CREDENTIAL2_SELECT   Select;
    EFI_CREDENTIAL2_DESELECT Deselect;
    EFI_CREDENTIAL2_DEFAULT  Default;
    EFI_CREDENTIAL2_GET_INFO GetInfo;
    EFI_CREDENTIAL2_GET_NEXT_INFO GetNextInfo;
    EFI_CREDENTIAL_CAPABILITIES Capabilities;
    EFI_CREDENTIAL2_DELETE   Delete;
};

extern EFI_GUID gEfiUserCredential2ProtocolGuid;
```

Example 241—User Credential Protocol

27

Load File Driver Design Guidelines

The Load File Protocol is used to support booting from a device type which does not fit cleanly into any of the standard device types supported by the *UEFI Specification*. A UEFI Boot Manager can only boot through the Simple File System Protocol or the Load File Protocol. If a device must be a boot device and cannot directly or indirectly produce the Simple File System Protocol or indirectly produce the Load File Protocol, then a Load File Protocol must be implemented. The indirect production of Simple File System and the Load File Protocol may not always be obvious. The EDK II provides a number of platform-agnostic drivers that help produce the Simple File System Protocol and the Load File Protocol through several layers of UEFI drivers. For example, a UEFI Driver that produces the Block I/O Protocol is sufficient to produce the Simple File System Protocol if the Disk I/O Driver, Partition Driver, and FAT File System Driver are also included in the platform. Review all the other boot device types described in this guide and the *UEFI Specification* before choosing to implement the Load File Protocol.

Note: *The Load File Protocol should not be implemented for any standard device type which has a defined driver hierarchy (e.g. USB, SCSI, and ATA).*

27.1 Assumptions

The rest of this chapter assumes that the Driver Checklist in [Chapter 2](#) has been followed and that the following items have already been identified:

- UEFI Driver Type
- Optional UEFI Driver features
- Supported CPU architectures
- Consumed protocols that are used to produce the User Credential Protocol.

UEFI drivers that produce the Load File Protocol typically follow the UEFI Driver Model. However, it is possible to implement UEFI drivers that directly produce the Load File Protocol for a single device in a platform. In this case a Root Bridge Driver implementation may be more appropriate than a UEFI Driver Model implementation.

27.2 Load File Protocol Implementation

The implementation of the Load File Protocol is typically found in the file `LoadFile.c`. [Appendix A](#) contains a template for a `LoadFile.c` file for a UEFI Driver. The list of tasks to implement the Load File Protocol is as follows:

- Add global variable for the `EFI_LOAD_FILE_PROTOCOL` instance to `LoadFile.c`.
- Implement the `LoadFile()` service in `LoadFile.c`.

The example below shows the protocol interface structure for the Load File Protocol for reference. This protocol is composed of a single service called `LoadFile()`. This service is typically used by a UEFI Boot Manager to boot a UEFI OS Loader or other UEFI Application from a device that does not directly or indirectly support the Simple File System Protocol.

```
typedef struct _EFI_LOAD_FILE_PROTOCOL EFI_LOAD_FILE_PROTOCOL;  
  
struct _EFI_LOAD_FILE_PROTOCOL {  
    EFI_LOAD_FILE LoadFile;  
};  
  
extern EFI_GUID gEfiLoadFileProtocolGuid;
```

Example 242—Load File Protocol

27.2.1 LoadFile() Implementation

The singular function `LoadFile()` of this protocol causes the driver to load the specified file from media into a buffer in system memory without the overlying layers knowing anything about the media that the file is stored on.

- Verify that the `FilePath` represents a file accessible by this device.
- Verify that the file specified by `FilePath` exists. If it does not exist, check `BootPolicy` to see if inexact `FilePath` is allowed.
- Verify that Buffer is large enough to return the entire file by examining `BufferSize` parameter. If not large enough, place correct size in `BufferSize` and return `EFI_BUFFER_TOO_SMALL`.

IPF Platform Porting Considerations

When writing a UEFI driver, there are steps that can be taken to help make sure the driver functions properly on an IPF platform. The guidelines listed in this chapter help improve the portability of UEFI drivers, and explain some of the pitfalls that may be encountered when a UEFI driver is ported to an IPF platform.

[Chapter 4](#) covers the general guidelines for implementing a UEFI Driver that is compatible with both 32-bit and 64-bit CPU architectures. If a 32-bit UEFI Driver is being ported to IPF, then make sure the guidelines from [Chapter 4](#) are followed. This chapter focuses on issues that are specific to IPF. In general, the guidelines for implementing a UEFI Driver for IPF are more rigorous than other CPU architectures. If a UEFI Driver is implemented and validated for IPF, then there is a good chance that the UEFI Driver can be easily ported to most of the other CPU architecture supported by the *UEFI Specification*.

In addition, the *DIG64 Specification* requires some protocols that are considered obsolete by the latest *UEFI Specification*. This means UEFI Drivers for IPF may have to produce some extra protocols from older versions of the *EFI Specification* and *UEFI Specification* in order to be conformant with the *DIG64 Specification*. The additional protocols are listed below. Other chapters of the guide provide recommendations on how to implement these protocols and this topic will not be covered further in this chapter.

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

28.1 General notes about porting to IPF platforms

When porting to IPF platform, most developers take as much code as possible that already exists and reuse it for the IPF platform. Unfortunately, some developers porting code do not rigorously follow the UEFI conventions, such as using only the data types defined in the Calling Conventions section of the *UEFI Specification*. Others may not follow best coding practices. This is a critical issue for IPF platforms because, although such code might work the first time, it may fail a more complete set of validation tests. It is also very likely that the code may not work when compiled with a different compiler, or after another developer performs maintenance on the code.

- Use data types defined by the Calling Conventions section of the *UEFI Specification*.
- Use compiler flag settings that guarantee that the UEFI calling conventions for IPF are followed.
- If a UEFI driver contains assembly language sources for a different CPU architecture, then those sources must be converted to either IPF assembly

language sources or to CPU agnostic C language sources. Conversion to C language sources is **recommended** and the EDK II library **BaseLib** and other EDK II libraries provide functions that may reduce or eliminate the need to assembly code in UEFI Drivers.

- Avoid alignment faults. This is the top issue in porting a UEFI driver to an IPF platform. Alignment faults may be due to type casting, packed data structures, or unaligned data structures.

28.2 Alignment Faults

The single most common issue with UEFI drivers for IPF platforms is alignment. Alignment faults cannot occur on IA32, X64, or EBC platforms, but can occur on IPF platforms. The IPF platform requires that all transactions 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 guarantees that accessing global variables, function parameters, local variables, and fields of data structures do not cause alignment faults.

Alignment faults can be generated when:

- C code can generate an alignment fault when a pointer is cast from one type to another or when packed data structures are used.
- Data structures are declared to be byte packed using `#pragma pack(1)` or equivalent.
- Assembly language can generate an alignment fault, but it is the assembly programmer's responsibility to ensure alignment faults are not generated. This topic is not covered further in this guide.

28.3 Casting Pointers

The example below shows an example that generates an alignment fault on an IPF platform. The first read access through `SmallValuePointer` is aligned because `LargeValue` is on a 64-bit boundary. However, the second read access though `SmallValuePointer` generates 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 is unaligned.

```
#include <Uefi.h>

UINT64 LargeValue;
UINT32 *SmallValuePointer;
UINT32 SmallValue;

SmallValuePointer = (UINT32 *)&LargeValue;

// 
// Works
//
SmallValue      = *SmallValuePointer;
```

```

SmallValuePointer = (UINT32 *)((UINT8 *)&LargeValue + 1);

//
// Fails. Generates an alignment fault
//
SmallValue      = *SmallValuePointer;

```

Example 243—Pointer-cast alignment fault

Example 244, below, shows the same example as Example 243, above, but has been modified to prevent the alignment fault. The second read access through `SmallValuePointer` is replaced with a call to the EDK II library **BaseLib** function called `ReadUnaligned32()` 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 all CPU architectures supported by the *UEFI Specification*.

```

#include <Uefi.h>
#include <Library/BaseLib.h>

UINT64 LargeValue;
UINT32 *SmallValuePointer;
UINT32 SmallValue;

SmallValuePointer = (UINT32 *)&LargeValue;

//
// Works
//
SmallValue      = *SmallValuePointer;

SmallValuePointer = (UINT32 *)((UINT8 *)&LargeValue + 1);

//
// Works
//
SmallValue      = ReadUnaligned32 (SmallValuePointer);

```

Example 244—Corrected pointer-cast alignment fault

EDK II library **BaseLib** provides several functions to help perform unaligned accessed in a safe manner. These functions perform a direct access on CPU architectures that do not generate alignment faults, and break the access up into small aligned pieces on CPU architectures that do generate alignment faults. The list of unaligned access functions from the EDK II library **BaseLib** includes the following:

- `ReadUnaligned64()`
- `ReadUnaligned32()`
- `ReadUnaligned24()`
- `ReadUnaligned16()`
- `WriteUnaligned64()`
- `WriteUnaligned32()`

- `WriteUnaligned24()`
- `WriteUnaligned16()`

28.4 Packed Structures

The following example shows another example that generates an alignment fault on an IPF platform. The first read access from `MyStructure.First` always works because the 8-bit value is always aligned. However, the second read access from `MyStructure.Second` always fails because the 32-bit value is never aligned on a 4-byte boundary.

```
#include <Uefi.h>

#pragma pack(1)
typedef struct {
    UINT8   First;
    UINT32  Second;
} MY_STRUCTURE;
#pragma pack()

MY_STRUCTURE  MyStructure;
UINT8          FirstValue;
UINT32         SecondValue;

// 
// Works
//
FirstValue = MyStructure.First;

//
// Fails. Generates an alignment fault
//
SecondValue = MyStructure.Second;
```

Example 245—Packed structure alignment fault

The next example shows the same example as Example 245, above, but has been modified to prevent the alignment fault. The second read access from `MyStructure.Second` is replaced with a call to the EDK II library **BaseLib** function called `ReadUnaligned32()` 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 all CPU architectures supported by the *UEFI Specification*.

```
#include <Uefi.h>
#include <Library/BaseLib.h>

#pragma pack(1)
typedef struct {
    UINT8   First;
    UINT32  Second;
} MY_STRUCTURE;
#pragma pack()

MY_STRUCTURE  MyStructure;
UINT8          FirstValue;
UINT32         SecondValue;
```

```

//  

// Works  

//  

FirstValue = MyStructure.First;  

//  

// Works  

//  

SecondValue = ReadUnaligned32 ((VOID *)&MyStructure.Second);

```

Example 246—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. The EDK II library `BaseMemoryLib` provides the `CopyMem()` service that handles unaligned copy operations, so an alignment fault is never 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 data structure is built and stored on a mass storage device or transmitted on a network.

28.5 UEFI Device Paths

The technique of using the EDK II library `BaseLib` functions to perform unaligned reads and writes is functional, but can become tedious if a large number of fields in data structures need to be accessed. In these 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 UEFI device path nodes and parsing network packets.

The device path nodes in a UEFI device path are packed together so they 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. UEFI device paths and UEFI device paths nodes may be passed around as opaque data structures, but whenever the fields of a UEFI device path node are 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. As each layer of a network packet is examined, the packet may need to be copied to a 64-bit aligned location before the individual fields of the packet are examined.

The following example shows an example of a function that parses a UEFI device path and extracts the 32-bit HID and UID from an ACPI device path node. This example generates an alignment fault if `DevicePath` is not aligned on a 32-bit boundary.

```

#include <Uefi.h>
#include <Protocol/DevicePath.h>

VOID
EFIAPI
GetAcpiHidUid (
    EFI_DEVICE_PATH_PROTOCOL *DevicePath,

```

```

    UINT32          *Hid,
    UINT32          *Uid
)
{
    ACPI_HID_DEVICE_PATH  *AcpiDevicePath;

    AcpiDevicePath = (ACPI_HID_DEVICE_PATH *)DevicePath;

    //
    // Wrong. May cause an alignment fault.
    //
    *Hid = AcpiDevicePath->HID;

    //
    // Wrong. May cause an alignment fault.
    //
    *Uid = AcpiDevicePath->UID;
}

```

Example 247—UEFI device path node alignment fault

Example 248, below, shows the corrected version of Example 247, above. 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 IPF platforms. The fields of the ACPI device path node can then be safely accessed without generating an alignment fault.

```

#include <Uefi.h>
#include <Protocol/DevicePath.h>
#include <Library/BaseMemoryLib.h>

VOID
EFIAPI
GetAcpiHidUid (
    EFI_DEVICE_PATH_PROTOCOL  *DevicePath,
    UINT32          *Hid,
    UINT32          *Uid
)
{
    ACPI_HID_DEVICE_PATH AcpiDevicePath;

    CopyMem (&AcpiDevicePath, DevicePath, sizeof (ACPI_HID_DEVICE_PATH));

    //
    // Correct. Guaranteed not to generate an alignment fault.
    //
    *Hid = AcpiDevicePath.HID;

    //
    // Correct. Guaranteed not to generate an alignment fault.
    //
    *Uid = AcpiDevicePath.UID;
}

```

Example 248—Corrected UEFI device path node alignment fault

28.6 PCI Configuration Header 64-bit BAR

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 in order to guarantee that an alignment fault is not generated.

The following two methods can be used to prevent an alignment fault when a 64-bit BAR value is extracted from a PCI configuration header:

- Use `ReadUnaligned64()` to read the BAR contents
- Use `CopyMem()` to transfer the BAR contents into a 64-bit aligned location.
- Collect the two 32-bit values that compose the 64-bit BAR, and combine them into a 64-bit value.

The example below shows the incorrect method of extracting a 64-bit BAR from a PCI configuration header, and then shows three correct methods.

```
#include <Uefi.h>
#include <IndustryStandard/Pci.h>
#include <Library/BaseMemoryLib.h>
#include <Library/BaseLib.h>

UINT64
EFIAPI
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.
    //
    BarValue = ReadUnaligned64 (BarPointer64);

    //
    // Correct. Guaranteed not to generate an alignment fault.
    //
    CopyMem (&BarValue, BarPointer64, sizeof (UINT64));

    //
    // Correct. Guaranteed not to generate an alignment fault.
    //
    BarValue = (UINT64)(*BarPointer32 | LShiftU64 (*(BarPointer32 + 1), 32));

    return BarValue;
}
```

Example 249—Accessing a 64-bit BAR in a PCI configuration header

28.7 Speculation and floating point register usage

IPF platforms support speculative memory accesses and a large number of floating point registers. UEFI drivers that are compiled for IPF platforms must follow the calling conventions defined in the *SAL Specification*. The *SAL Specification* only allows 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 UEFI pre-boot environment. These requirements mean that the correct compiler and linker switches must be set correctly to guarantee that these calling conventions are followed. The EDK II provides proper compiler and linker settings for several tool chains that support IPF platforms. These settings may have to be adjusted if updates to a tool chain are release or if a different tool chain is used. The following table shows the compiler flags for a few different compilers. The compiler flag that specifies that only the first 32 floating point registers may be used for Microsoft® compilers is `/QIPF_fr32`. The equivalent flag of Intel compilers is `/QIA64_fr32`.

EFI Byte Code Porting Considerations

There are a few considerations to keep in mind when writing drivers that may be ported to EBC (EFI byte code). 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, the C code may not require any changes. In this case, a native driver may be ported to EBC simply by recompiling the driver sources using the Intel® C Compiler for EFI Byte Code. The tasks required to convert a UEFI Driver to an EBC include the following:

- Port assembly language sources to C language sources.
- Port C++ language sources to C language sources.
- Eliminate use of the float type.
- Convert floating point math operations to integer math operations.
- Eliminate use of sizeof() in statements that require a constant.
- Avoid arithmetic operations and comparisons between natural integers and fixed size integers. Some specific combinations produce unexpected results.
- Optimize for performance

29.1 No Assembly Support

The only tools that are provided with the Intel® C Compiler for EFI Byte Code are a C compiler and a linker. No assemblers for EBC are provided. The lack of an EBC assembler is by design, because the EBC instruction set is optimized for a C compiler. If a UEFI Driver is being ported to EBC, all assembly language sources for 32-bit and 64-bit processors must be ported to C language sources.

29.2 No C++ Support

The Intel® C Compiler for EFI Byte Code does not support C++. If there is any C++ code in a UEFI driver being ported to EBC, then that C++ language sources must be converted to C language sources.

29.3 No Floating Point Support

There is no floating-point support in the EBC virtual machine, which means that the type `float` is not supported by the Intel® C Compiler for EFI Byte Code. If a UEFI Driver is being ported to EBC and the UEFI Driver uses floating-point math, then the UEFI Driver must be converted to use fixed-point math based on integer operands and operators.

29.4 Use of sizeof()

In some cases, `sizeof()` is computed at runtime for EBC code, whereas `sizeof()` is never computed at runtime for native code. Because pointers and the UEFI data types `INTN` and `UINTN` are different sizes on different CPU architectures, an EBC image must adapt to the platform on which it is executing. The example below shows several examples of simple and complex data types. For the types that return different sizes for 32-bit versus 64-bit processors, the EBC compiler generates code that computes the correct values at runtime when executing on 32-bit and 64-bit processors.

```
#include <Uefi.h>

typedef enum {
    Red,
    Green,
    Blue
} COLOR_TYPE;

#pragma pack(1)
typedef struct {
    UINT64  ValueU64;
    UINT32  ValueU32;
    UINT16  ValueU16;
    UINT8   ValueU8;
} FIXED_STRUCTURE;

typedef struct {
    UINTN   ValueUN;
    VOID    *Pointer;
    UINT64  ValueU64;
    UINT32  ValueU32;
} VARIABLE_STRUCTURE;
#pragma pack()

UINT64  Size;

Size = sizeof (UINT64);           // 8 bytes on all CPUs
Print (L"Size = %d\n", Size);
Size = sizeof (UINT32);           // 4 bytes on all CPUs
Print (L"Size = %d\n", Size);
Size = sizeof (UINT16);           // 2 bytes on all CPUs
Print (L"Size = %d\n", Size);
Size = sizeof (UINT8);            // 1 byte on all CPUs
Print (L"Size = %d\n", Size);
Size = sizeof (UINTN);            // 4 bytes on 32-bit CPU, 8 bytes on 64-bit CPU
Print (L"Size = %d\n", Size);
Size = sizeof (INTN);             // 4 bytes on 32-bit CPU, 8 bytes on 64-bit CPU
Print (L"Size = %d\n", Size);
Size = sizeof (COLOR_TYPE);       // 4 bytes on 32-bit CPU, 8 bytes on 64-bit CPU
Print (L"Size = %d\n", Size);
Size = sizeof (VOID *);           // 4 bytes on 32-bit CPU, 8 bytes on 64-bit CPU
Print (L"Size = %d\n", Size);

//
// 15 bytes on 32-bit CPU, 15 bytes on 64-bit CPU
//
Size = sizeof (FIXED_STRUCTURE);
Print (L"Size = %d\n", Size);

//
// 20 bytes on 32-bit CPU, 28 bytes on 64-bit CPU
//
Size = sizeof (VARIABLE_STRUCTURE);
Print (L"Size = %d\n", Size);
```

Example 250—Size of data types with EBC

29.4.1 Global Variable Initialization

In a native compile the value of `sizeof (UINTN)` is computed 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. Instead, it generates code to calculate this value at execution time if the result is different on different CPU architectures. This limitation means that EBC code cannot use `sizeof (UINTN)`, `sizeof (INTN)`, and `sizeof (VOID *)` (or other pointer types) in C language statements that require constant expressions.

Note: *The type `EFI_STATUS` is required to be type `UINTN` by the UEFI Specification. This means that a variable of type `EFI_STATUS` cannot be used in C language statements that require constant expressions.*

The code in the following example **fails** when compiled for EBC.

```
#include <Uefi.h>
#include <UefiBootServicesTableLib.h>

//
// Global variable definitions
//
UINTN      IntegerSize = sizeof (UINTN);           // EBC compiler error
UINTN      PointerSize = sizeof (VOID*);            // EBC compiler error
EFI_STATUS Status      = EFI_INVALID_PARAMETER;    // EBC compiler error
```

Example 251—Global Variable Initialization that fails for EBC

The following example shows one method to address the EBC compiler errors in the previous example. The general technique is to move the initialization of global variables that are impacted by the EBC specific issue into the driver entry point or other function that executes before the global variables are used.

```
#include <Uefi.h>
#include <UefiBootServicesTableLib.h>

//
// Global variable definition
//
UINTN      IntegerSize;
UINTN      PointerSize;
EFI_STATUS Status;

VOID
InitializeGlobals (
  VOID
)
{
  IntegerSize = sizeof (UINTN);
  PointerSize = sizeof (VOID*);
  Status      = EFI_INVALID_PARAMETER;
}
```

Example 252—Global Variable Initialization that works for EBC

29.4.2 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. UEFI status codes values such as `EFI_SUCCESS` and `EFI_UNSUPPORTED` are defined differently on different CPU architectures. As a result, UEFI status codes cannot be used in `case` expressions. The following example shows examples using `case` statements.

```
#include <Uefi.h>

UINTN Value;

switch (Value) {
    case 0: // Works because 0 is a constant.
        break;
    case sizeof (UINT16): // Works because sizeof (UINT16) is always 2.
        break;
    case sizeof (UINTN): // EBC compiler error. sizeof (UINTN) is not constant.
        break;
    case EFI_UNSUPPORTED: // EBC compiler error. EFI_UNSUPPORTED is not constant.
        break;
}
```

Example 253—Case statements that fail for EBC

One solution to this issue is to convert `case` statements into `if/else` statements. The example below shows the equivalent functionality as Example 253, above, but does not generate any EBC compiler errors.

```
#include <Uefi.h>

UINTN Value;

switch (Value) {
    case 0: // Works because 0 is a constant.
        break;
    case sizeof (UINT16): // Works because sizeof (UINT16) is always 2.
        break;
}
if (Value == sizeof (UINTN)) {
} else if (Value == EFI_UNSUPPORTED) {
```

Example 254—Case statements that work for EBC

29.5 Natural Integers and Fixed Size Integers

UEFI Drivers should only use the integer data types defined in the Calling Conventions section of the *UEFI Specification*. Even when this recommendation is followed, there is an additional limitation of the EBC architecture. UEFI Drivers with arithmetic calculations and comparisons between following integer types must be avoided:

- `INTN` and `UINT8`
- `INTN` and `UINT16`
- `INTN` and `UINT32`
- `UINTN` and `INT64`

29.6 Memory ordering

The EBC architecture is required to be strongly ordered, and the EBC virtual machine interpreter ensures that all memory transactions are strongly ordered. The EDK II includes a complete implementation of the EBC virtual machine interpreter in the `MdeModulePkg` in the directory `MdeModulePkg/Universal/EbcDxe`.

EBC drivers are not required to use the EDK II library `BaseLib` function `MemoryFence()` when strong ordering is required. However, UEFI Drivers compiled for other CPU architectures may require the use of the `MemoryFence()` function to enforce strong ordering. The EDK II library `BaseLib` implementation of `MemoryFence()` for EBC is an empty function. This means there is no performance penalty for `MemoryFence()` calls in UEFI Drivers compiled for EBC.

29.7 Performance considerations

All EBC executables require an EBC virtual machine interpreter to be executed. Because all EBC executables are running through an interpreter, they execute slower than native UEFI executables. As a result, a UEFI driver that is compiled with an EBC compiler should be optimized for performance to improve the usability of the UEFI Driver. [Chapter 4](#) covers speed optimization techniques that may be used to improve the performance of all UEFI Drivers.

The simplest way to maximize the speed of a UEFI Driver compiled for EBC is to maximize the use of UEFI Boot Services, UEFI Runtime Services, and protocols produced by other UEFI components. These calls outside of the UEFI Driver compiled for EBC help improve performance because those other services may be native calls that can be executed without the overhead of the EBC virtual machine interpreter. If all UEFI Drivers compiled for EBC follow the recommendation, even if one UEFI Driver compiled for EBC calls another UEFI Driver compiled for EBC, the overhead of the EBC interpreter is still minimized.

29.7.1 Performance considerations for data types

Avoid declaration and initialization of variables or structures that contain native length data types such as `INTN`, `UINTN`, and pointers. One of the issues with initializing variables occurs during optimization. If variables are initialized statically, the compiler optimizes them for size and, for example, gives the variable a 32-bit placement or a 16-bit placement. This can create problems if the variables are a size that is different on different CPU architectures.

TIP: Initialize variables separately from declaring them.

The amount of variable initialization that is performed during EBC runtime initialization can be determined by viewing the PE/COFF sections of a UEFI Driver compiled for EBC. The linker provided with Microsoft™ tools provides a method to perform this operation. The command is:

```
link -dump -headers <filename>
```

This command dumps the different parts of an .EFI file. The goal is to minimize the _VARBSS_ section while maximizing the .data and .rdata sections of the PE/COFF image.

29.8 UEFI Driver Entry Point

The entry point to an EBC compiled image is a function is always called `EfiStart()`. This is the function that is shown as the entry point in the PE/COFF image that is produced by an EBC compile/link operation. The `EfiStart()` function performs the EBC runtime initialization that may vary from one UEFI Driver to another. At the end of the EBC runtime initialization, the function `EfiMain()` is called. The EDK II build system and libraries take care of these details, so a UEFI Driver implementation never contains functions with these names. In fact, the symbols `EfiStart()` and `EfiMain()` must be considered reserved, and cannot be used as function names or variable names in any UEFI driver implementation that is compiled for EBC.

The INF file for a UEFI Driver declares the C entry point in the `[Defines]` section in a define called `ENTRY_POINT`. All UEFI Drivers are linked to the EDK II library instance from the `MdePkg` called `UefiDriverEntryPoint`, and the `UefiDriverEntryPoint` library instance is responsible for calling the library constructors for all the libraries that a UEFI Driver is using either directly or indirectly. Once all the library constructors have been called, control is transferred to the `ENTRY_POINT` function defined in the INF file. This is where the C sources for a UEFI Driver implementation begin and the driver specific initialization is performed.

The sequence of calls in a UEFI Driver entry point compiled for EBC is as follows:

- `EfiStart()` – PE/COFF entry point that performs the required EBC runtime initialization. Calls `EfiMain()`.
- `EfiMain()` – Calls `_ModuleEntryPoint()`
- `_ModuleEntryPoint()` – Calls EDK II library constructors. Calls `ENTRY_POINT` function defined in INF file.
- `ENTRY_POINT` function – Performs UEFI Driver specific initialization.

Knowledge of this specific sequence of calls is not typically required by a UEFI Driver developer because it is very rare for anything to go wrong in `EfiStart()`, `EfiMain()` or `_ModuleEntryPoint()` functions. However, if a UEFI Driver compiled for EBC is being debugged, it is important to know that these extra actions do occur between the entry point of the PE/COFF image and the first line of C source code in the UEFI Driver implementation.

30

Building UEFI Drivers

This chapter provides an overview of how to compile and link a UEFI Driver in an EDK II build environment to produce a UEFI conformant UEFI Driver image that may be loaded and executed on a UEFI conformant platform. The steps required include:

- Create an EDK II package, if required, for the UEFI Driver
- Create directory for UEFI Driver in an existing EDK II package.
- Add INF file and all source files to UEFI Driver directory.
- Add file path to INF file to EDK II package DSC file.
- Build UEFI Driver using the EDK II build tool called **build.exe**.
- Locate UEFI Driver in the build output directory specified by DSC file.

For detailed information, refer to the EDK II Build Specification on
www.tianocore.org

30.1 Prerequisites

Before a UEFI Driver can be built, an EDK II build environment must be established on a development system. The EDK II project is maintained on <http://www.tianocore.org> and validated releases of the EDK II project are periodically posted. The current release is the UDK2010. It is **recommended** that a validated release of UDK2010 be used for UEFI Driver development instead of the trunk of the EDK II project because the trunk of the EDK II project is under active development. The UDK2010 release page includes links to documentation to help setup a build environment on a development system. Verify that one of the standard platforms builds correctly before proceeding.

In most cases, building a UEFI Driver only requires a few directories from an EDK II build environment. These include:

- **BaseTools** – Contains EDK II build tools
- **Conf** – Contains configuration files for EDK II build tools and supported compilers and linkers
- **MdePkg** – Contains the include files and libraries to support industry standard specifications. This content includes all of the published *UEFI Specifications* and *EFI Specifications* as well as includes files for industry standard buses such as PCI, USB, and SCSI.
- **MdeModulePkg** – Contains UEFI Drivers that can be used as reference. Also contains HII related libraries that may be used by UEFI Drivers that produce HII packages.

- **OptionRomPkg** – Sample package with three UEFI Drivers and a UEFI Application that can be used as a template for a device manufacturer's own package for UEFI Driver development.
- **MyDriverPkg** – Example EDK II package that contains the UEFI Drivers implemented by a device manufacturer. This directory name is only used for discussion purposes. Device manufacturers may generate their own directory name for their own package and may generate more than one package for their UEFI Driver content if required.

30.2 Create EDK II Package

The first step is to make sure there is an EDK II package available to which a new UEFI Driver can be added. If an EDK II package has already been created for UEFI Driver work, then this step may be skipped. Otherwise the following steps are required:

- Create a new directory that is a peer to **MdePkg** (e.g. **MyDriverPkg**).
- Create a subdirectory called **Include** (e.g. **MyDriverPkg/Include**).
- Create a subdirectory of **Include** called **Protocol** (e.g. **MyDriverPkg/Include/Protocol**).
- Create a subdirectory of **Include** called **Guid** (e.g. **MyDriverPkg/Include/Guid**).
- Create a subdirectory of **Include** called **Library** (e.g. **MyDriverPkg/Include/Library**).
- Add DEC file to the new package directory (e.g. **MyDriverPkg/MyDriverPkg.dec**).
- Add DSC file to the new package directory (e.g. **MyDriverPkg/MyDriverPkg.dsc**).

The following example shows an example directory structure for an EDK II WORKSPACE after creating the new package called **MyDriverPkg** following the steps listed above. The **Include** subdirectory is a place holder in case new Protocols, GUIDs, or Library Classes are required to support new UEFI Driver implementations.

```
BaseTools/
Conf/
MdePkg/
MdeModulePkg/
OptionRomPkg/
MyDriverPkg/
  MyDriverPkg.dec
  MyDriverPkgf.dsc
  Include/
    Protocol/
    Guid/
    Library/
```

Example 255—EDK II Package Directory

The [following example](#) shows an example DEC file **MyDriverPkg/MyDriverPkg.dec**. Every new DEC file must have a unique GUID value and name.

```
[Defines]
DEC_SPECIFICATION      = 0x00010005
PACKAGE_NAME            = MyDriverPkg
PACKAGE_GUID             = E972EFA5-75CC-4ade-A719-60DD9AE5217B
PACKAGE_VERSION           = 0.10

[Includes]
Include
```

Example 256—EDK II Package DEC File

The example below shows an example DSC file `MyDriverPkg/MyDriverPkg.dsc`. Every new DSC must have a unique `PLATFORM_GUID` value, `PLATFORM_NAME` and `OUTPUT_DIRECTORY` path. This DSC file example also contains the library mapping required to build a UEFI conformant UEFI Driver. Many other library mappings are possible with the content from the EDK II project, but many of this mappings use services that are not defined by the *UEFI Specification*, so the use of alternate mapping may produce a UEFI Driver that runs correctly on some platforms but not others.

```
[Defines]
PLATFORM_NAME          = MyDriverPkg
PLATFORM_GUID           = 7C297DD4-65D9-4dfe-B609-94330E607888
PLATFORM_VERSION         = 0.10
DSC_SPECIFICATION       = 0x00010005
OUTPUT_DIRECTORY         = Build/MyDriverPkg
SUPPORTED_ARCHITECTURES = IA32|IPF|X64|EBC|ARM
BUILD_TARGETS            = DEBUG|RELEASE
SKUID_IDENTIFIER          = DEFAULT

[LibraryClasses]
UefiDriverEntryPoint|MdePkg/Library/UefiDriverEntryPoint/UefiDriverEntryPoint.inf
UefiApplicationEntryPoint|MdePkg/Library/UefiApplicationEntryPoint/UefiApplicationEntryPoint.inf
UefiBootServicesTableLib|MdePkg/Library/UefiBootServicesTableLib/UefiBootServicesTableLib.inf
UefiLib|MdePkg/Library/UefiLib/UefiLib.inf
UefiRuntimeServicesTableLib|MdePkg/Library/UefiRuntimeServicesTableLib/UefiRuntimeServicesTableLib.inf
UefiRuntimeLib|MdePkg/Library/UefiRuntimeLib/UefiRuntimeLib.inf
MemoryAllocationLib|MdePkg/Library/UefiMemoryAllocationLib/UefiMemoryAllocationLib.inf
DevicePathLib|MdePkg/Library/UefiDevicePathLib/UefiDevicePathLib.inf
UefiUsbLib|MdePkg/Library/UefiUsbLib/UefiUsbLib.inf
UefiScsiLib|MdePkg/Library/UefiScsiLib/UefiScsiLib.inf
BaseLib|MdePkg/Library/BaseLib/BaseLib.inf
BaseMemoryLib|MdePkg/Library/BaseMemoryLib/BaseMemoryLib.inf
SynchronizationLib|MdePkg/Library/BaseSynchronizationLib/BaseSynchronizationLib.inf
PrintLib|MdePkg/Library/BasePrintLib/BasePrintLib.inf
DebugLib|MdePkg/Library/UefiDebugLibStdErr/UefiDebugLibStdErr.inf
DebugPrintErrorLevelLib|MdePkg/Library/BaseDebugPrintErrorLevelLib/BaseDebugPrintErrorLevelLib.inf
PostCodeLib|MdePkg/Library/BasePostCodeLibPort80/BasePostCodeLibPort80.inf
PcdLib|MdePkg/Library/BasePcdLibNull/BasePcdLibNull.inf
```

Example 257—EDK II Package DSC File

30.3 Create UEFI Driver Directory

The next step is to create a subdirectory in an EDK II package for the UEFI Driver contents including an INF file and all source files required to build the UEFI Driver. There are no restrictions on the directory structure organization within an EDK II package. The examples shown here are simple and only use one layer of directories. The `MdeModulePkg` is an example of an EDK II package with about 100 UEFI Drivers and

a more complex directory structure to organize the UEFI Drivers based on the protocol they consume and the features they provide.

[Appendix A](#) contains a template for an INF file for a UEFI Driver and a UEFI Runtime Driver. This template should be sufficient for most UEFI Driver implementations. The EDK Build Specifications on <http://www.tianocore.org> provide the full description of INF files and their supported syntax for describing all the packages, sources, library classes, protocols, and GUIDs required to compile and link a UEFI Driver.

The example below shows an example directory structure for an EDK II WORKSPACE after creating the new package called **MyDriverPkg** following the steps listed above and creating a subdirectory called **MyDriver** and adding an INF file and a C source file.

```
BaseTools/
Conf/
MdePkg/
MdeModulePkg/
OptionRomPkg/
MyDriverPkg/
  MyDriverPkg.dec
  MyDriverPkf.dsc
  Include/
    Protocol/
    Guid/
    Library/
  MyDriver/
    MyDriver.inf
    MyDriver.c
```

Example 258—UEFI Driver Directory

The following example shows an example INF file **MyDriverPkg/MyDriver/MyDriver.inf**. Every new INF must have a unique **FILE_GUID** value and **BASE_NAME**. This INF file example only uses the services from a single library class called **UefiDriverEntryPoint**. Every UEFI Driver must use this Library Class. Examples in earlier chapters show more complex driver examples that use more library classes. The DSC file in the previous section contains a mapping for the **UefiDriverEntryPoint** library and that mapping is to **MdePkg/Library/UefiDriverEntryPoint/UefiDriverEntryPoint.inf**.

```
[Defines]
  INF_VERSION      = 0x00010005
  BASE_NAME        = MyDriver
  FILE_GUID        = 1C0D95A7-C0D6-4054-9245-8E2C81FC9ECD
  MODULE_TYPE      = UEFI_DRIVER
  VERSION_STRING   = 1.0
  ENTRY_POINT      = MyDriverEntryPoint

[Sources]
  MyDriver.c

[Packages]
  MdePkg/MdePkg.dec

[LibraryClasses]
  UefiDriverEntryPoint
```

Example 259—UEFI Driver INF File

This example shows an example C source file `MyDriverPkg/MyDriver/MyDriver.c` that does not do anything other than just return `EFI_SUCCESS`.

```
#include <Uefi.h>

EFI_STATUS
EFIAPI
MyDriverEntryPoint (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    return EFI_SUCCESS;
}
```

Example 260—UEFI Driver C Source File

30.3.1 Disk I/O Driver Example

A more complete example of a simple UEFI Driver that follows the UEFI Driver Model is the Disk I/O driver in the MdeModulePkg. The directory path and INF file and source files are shown in the example below.

```
MdeModulePkg\
Universal\
Disk\
DiskIoDxe\
ComponentName.c
Diskio.c
Diskio.h
DiskIoDxe.inf
```

Example 261—Disk I/O UEFI Driver Source Files

30.3.2 Reserved Directory Names

The UEFI Drivers in the two examples above do not contain any instruction set architecture-specific files. This absence means that this driver is designed to be portable between all CPU architecture supported by the *UEFI Specification*. If a UEFI Driver requires instruction set architecture-specific source files, then those source files are typically placed in subdirectories below the UEFI driver's main directory in an EDK II package. A separate subdirectory is required for each instruction set architecture that the UEFI Driver supports. The table below lists the directory names that are reserved for the instruction set architecture-specific files.

Table 39—Reserved directory names

Directory name	Notes
Ia32	May contain <code>.c</code> , <code>.h</code> , and <code>.asm</code> files.
Ipf	May contain <code>.c</code> , <code>.h</code> , and <code>.s</code> files.
Ebc	May contain <code>.c</code> and <code>.h</code> files.
X64	May contain <code>.c</code> , <code>.h</code> , and <code>.asm</code> files.

Note: Code written in EBC is still C language code, and the sources look the same as for any other driver. It is when the compiler converts it from a high-level language (C) to object code (EBC versus native code) that the difference becomes evident.

For EBC, the object code generated is not native to the processor but rather is pseudo-object-code that looks like RISC processor object code. That code is fed into an interpreter, which interprets each instruction and acts upon it. The EBC output is not 32-bit or 64-bit-based, but rather conforms to its own standard. Thus a system with a valid interpreter for its architecture can translate EBC compiled code into operational instructions on any given architecture.

30.3.3 EBC Virtual Machine Driver Example

The following example shows the directory structure for another driver that includes instruction set architecture-specific files for three of the supported instruction sets. The files `EbcLowLevel.asm` and `EbcLowLevel.s` contain logic that must be implemented in assembly to handle the transitions between native execution and the EBC interpreter. 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, a UEFI driver should be implemented in C with no instruction set architecture-specific files, which reduces the development time, reduces maintenance costs, and increases portability.

```
MdeModulePkg\
  Universal\
    EbcDxe\
      EbcDxe.inf
      EbcExecute.c
      EbcExecute.h
      EbcInt.c
      EbcInt.h
    Ia32\
      EbcSupport.c
      EbcLowLevel.asm
      EbcLowLevel.S
    Ipf\
      EbcSupport.c
      EbcSupport.h
      EbcLowLevel.s
    X64\
      EbcSupport.c
      EbcLowLevel.asm
      EbcLowLevel.S
```

Example 262—EBC driver with instruction set architecture-specific files

30.4 Adding a UEFI Driver to DSC File

The list of UEFI Drivers that need to be built must be added to the `[Components]` section of a DSC file. Once a UEFI Driver has been added to the `[Components]` section, an

attempt is made to build the UEFI Driver every time the EDK II build tool called **build.exe** is invoked with the appropriate parameters.

This example shows the same example DSC file **MyDriverPkg/MyDriverPkg.dsc**, except it has now been updated to list **MyDriverPkg/MyDriver/MyDriver.inf** in the **[Components]** section.

```
[Defines]
PLATFOM_NAME           = MyDriverPkg
PLATFOM_GUID            = 7C297DD4-65D9-4dfe-B609-94330E607888
PLATFOM_VERSION          = 0.10
DSC_SPECIFICATION        = 0x00010005
OUTPUT_DIRECTORY          = Build/MyDriverPkg
SUPPORTED_ARCHITECTURES   = IA32|IPF|X64|EBC|ARM
BUILD_TARGETS              = DEBUG|RELEASE
SKUID_IDENTIFIER           = DEFAULT

[LibraryClasses]
UefiDriverEntryPoint|MdePkg/Library/UefiDriverEntryPoint/UefiDriverEntryPoint.inf
UefiApplicationEntryPoint|MdePkg/Library/UefiApplicationEntryPoint/UefiApplicationEntryPoint.inf
UefiBootServicesTableLib|MdePkg/Library/UefiBootServicesTableLib/UefiBootServicesTableLib.inf
UefiLib|MdePkg/Library/UefiLib/UefiLib.inf
UefiRuntimeServicesTableLib|MdePkg/Library/UefiRuntimeServicesTableLib/UefiRuntimeServicesTableLib.inf
UefiRuntimeLib|MdePkg/Library/UefiRuntimeLib/UefiRuntimeLib.inf
MemoryAllocationLib|MdePkg/Library/UefiMemoryAllocationLib/UefiMemoryAllocationLib.inf
DevicePathLib|MdePkg/Library/UefiDevicePathLib/UefiDevicePathLib.inf
UefiUsbLib|MdePkg/Library/UefiUsbLib/UefiUsbLib.inf
UefiScsiLib|MdePkg/Library/UefiScsiLib/UefiScsiLib.inf
BaseLib|MdePkg/Library/BaseLib/BaseLib.inf
BaseMemoryLib|MdePkg/Library/BaseMemoryLib/BaseMemoryLib.inf
SynchronizationLib|MdePkg/Library/BaseSynchronizationLib/BaseSynchronizationLib.inf
PrintLib|MdePkg/Library/BasePrintLib/BasePrintLib.inf
DebugLib|MdePkg/Library/UefiDebugLibStdErr/UefiDebugLibStdErr.inf
DebugPrintErrorLevelLib|MdePkg/Library/BaseDebugPrintErrorLevelLib/BaseDebugPrintErrorLevelLib.inf
PostCodeLib|MdePkg/Library/BasePostCodeLibPort80/BasePostCodeLibPort80.inf
PcdLib|MdePkg/Library/BasePcdLibNull/BasePcdLibNull.inf

[Components]
MyDriverPkg/MyDriver/MyDriver.inf
```

Example 263—EDK II Package DSC File

30.5 Building a UEFI driver

Building a UEFI Driver involves the use of the **build.exe** command provided with the EDK II tools. If the pre-requisites were followed at the beginning of this chapter, then the only flag that need to be passed into **build.exe** is the DSC file that is to be used for the build.

```
build -p MyDriverPkg/MyDriverPkg.dsc
```

If the build completes successfully, then the UEFI Driver generated can be found in the build output directory that is specified in the DSC file. In the example above, **OUTPUT_DIRECTORY** is set to **Build/MyDriverPkg**. The [following example](#) shows where **MyDriver.efi** is located. This specific example shows that a **DEBUG** build was used with a Microsoft family compiler to generate **MyDriver.efi** for IA32.

```
Build\  
  MyDriverPkg\  
    DEBUG_VS2005x86\  
      IA32\  
        MyDriver.efi
```

Example 264—Build Output Directory

If a UEFI Driver needs to be built as a **DEBUG** build or a **RELEASE** build, this can be specified on the command line. The following two examples show how to build for **DEBUG** and **RELEASE**. If the **-b** flag is not specified, then the build type is retrieved from **Conf/target.txt**.

```
build -b DEBUG -p MyDriverPkg/MyDriverPkg.dsc  
build -b RELEASE -p MyDriverPkg/MyDriverPkg.dsc
```

If a UEFI Driver needs to be built for other CPU architectures, then those can also be specified on the command line. The following 4 examples show how to build for IA32, X64, IPF, and EBC if the compiler and linkers installed support all these architectures.

```
build -a IA32 -p MyDriverPkg/MyDriverPkg.dsc  
build -a X64 -p MyDriverPkg/MyDriverPkg.dsc  
build -a IPF -p MyDriverPkg/MyDriverPkg.dsc  
build -a EBC -p MyDriverPkg/MyDriverPkg.dsc
```

The 4 separate commands above can also be combined into a single command.

```
build -a IA32 -a X64 -a IPF -a EBC -p MyDriverPkg/MyDriverPkg.dsc
```

The EDK II also supports a configuration file for builds in the file path **Conf/target.txt**. This file may be updated with the specific configuration that is most commonly used. For example, the **ACTIVE_PLATFORM** can be set to **MyDriverPkg/MyDriverPkg.dsc**, and the build command can then be invoked with no parameters at all.

```
build
```

Please see the EDK II User's Manual and other EDK II documents for more details on how to use the build command and for details on INF files, DEC files, and DSC files.

31

Testing and Debugging UEFI Drivers

This chapter includes some best practices for testing a debugging UEFI Drivers that should help minimize production issues and simplify debugging. The most common tool used to do initial testing of a UEFI Driver is the UEFI Shell. Once the basic functionality is established, more rigorous testing can be performed. At a minimum a UEFI Driver should be tested with the following scenarios. This chapter focuses on use of the UEFI Shell and method of augmenting UEFI Drivers to improve debug ability.

- Use UEFI Shell for basic functionality and debug
- Run UEFI Self Certification Tests available from <http://www.uefi.org>
- Install UEFI Operating Systems
- Boot UEFI operating Systems

31.1 Native and EBC

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 are fewer timing sensitivities to the driver, and it is more robust.

31.2 Compiler Optimizations

When optimization is enabled, the code being debugged is different than the code debugged without optimization. For example, some instructions might be more efficient for the processor when optimization is turned on but they may introduce timing issues.

TIP: Disable compiler optimizations during the development and debugging phases.

The example below shows the **[BuildOptions]** section that may be added to the DSC example from [Chapter 30](#) to disable compiler optimizations for all compilers.

```
[BuildOptions]
GCC: *_*_CC_FLAGS = -O0
INTEL: *_*_CC_FLAGS = /Od
MSFT: *_*_CC_FLAGS = /Od
```

Example 265—EDK II Package DSC File with Optimizations Disabled

31.3 UEFI Shell Debugging

There are several UEFI Shell commands that can be used to help debug UEFI drivers. These UEFI Shell commands are documented in the *EFI Shell Users Guide*.

Caution: *The UEFI Shell that is included in EDK II is a reference implementation of a 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.*

A detailed description of a UEFI Shell commands may be displayed by using the built-in UEFI Shell `help` command. The following table lists UEFI Shell commands that may be useful when testing and debugging UEFI drivers along with the protocol and/or service exercised. Type `shell -h` to get a list of all available shell commands.

31.3.1 Testing Specific Protocols

The following table lists Shell commands that might be useful in testing specific protocols.

Table 40—UEFI Shell commands

Command	Protocol tested	Service tested
Load –nc	Driver Binding	Driver entry point. Supported()
Load	Driver Binding	Driver entry point. Supported()
	Driver Binding	Start()
Unload	Loaded Image	Unload()
Connect	Driver Binding	Supported()
	Driver Binding	Start()
Disconnect	Driver Binding	Stop(). Note: The UEFI driver must specifically disconnect (destroy) all child handles and device paths associated with the handle for the driver being stopped.
Reconnect	Driver Binding	Supported()
	Driver Binding	Start()
	Driver Binding	Stop()
Drivers	Component Name and Component Name2	GetDriverName()
Devices	Component Name and Component Name2	GetDriverName() GetControllerName()
DevTree	Component Name and Component Name2	GetControllerName()

Command	Protocol tested	Service tested
Dh -d	Component Name and Component Name2	GetDriverName() GetControllerName()
DrvCfg -s	Driver Configuration and Driver Configuration 2	This command used to test the SetOptions() service. These protocols are supported by EFI 1.10 and UEFI 2.0.
DrvCfg -f	Driver Configuration and Driver Configuration 2	This command used to test the ForceDefaults() service. These protocols are supported by EFI 1.10 and UEFI 2.0.
DrvCfg -v	Driver Configuration and Driver Configuration 2	This command used to test the OptionsValid() service. These protocols are supported by EFI 1.10 and UEFI 2.0.
DrvDiag	Driver Diagnostics and Driver Diagnostics2	RunDiagnostics()

31.3.2 Other Testing

There are other tests that can be performed from within the UEFI shell. These are not testing a specific protocol, but are testing functionality and for other coding practices.

Table 41—Other Shell Testing Procedures

Shell command sequence	What it tests
<pre>Shell> Memmap Shell> Dh Shell> Load DriverName.efi Shell> Memmap Shell> Dh Shell> Unload DriverHandle Shell> Memmap Shell> Dh</pre>	Tests for incorrectly matched up DriverEntryPoint and Unload() functions. This catches memory allocation that is not unallocated, and catches protocols that are installed and not uninstalled, etc...
<pre>Shell> Memmap Shell> Connect DeviceHandle DriverHandle Shell> Memmap Shell> Disconnect DeviceHandle DriverHandle Shell> Memmap Shell> Reconnect DeviceHandle Shell> Memmap</pre>	Tests for incorrectly matched up Driver Binding Start() and Stop() functions. This catches memory allocation that is not unallocated.
<pre>Shell> dh Shell> Connect DeviceHandle DriverHandle Shell> dh Shell> Disconnect DeviceHandle DriverHandle Shell> dh Shell> Reconnect DeviceHandle Shell> dh</pre>	Tests for incorrectly matched up Driver Binding Start() and Stop() functions. This catches protocols that are installed and not uninstalled.

<pre> Shell> OpenInfo DeviceHandle Shell> Connect DeviceHandle DriverHandle Shell> OpenInfo DeviceHandle Shell> Disconnect DeviceHandle DriverHandle Shell> OpenInfo DeviceHandle Shell> Reconnect DeviceHandle Shell> OpenInfo DeviceHandle </pre>	Tests for incorrectly matched up Driver Binding <code>Start()</code> and <code>Stop()</code> functions. This catches protocols that are opened and not closed.
--	--

31.3.3 Loading UEFI drivers

The following table lists the UEFI Shell commands that are available to load and start UEFI drivers.

Table 42—UEFI Shell commands for loading UEFI drivers

Command	Description
Load	<p>Loads a UEFI driver from a file. UEFI driver files typically have an extension of <code>.efi</code>. The <code>Load</code> command has one important option, the <code>-nc</code> (“No Connect”) option, for UEFI driver developers. When the <code>Load</code> command is used without the <code>-nc</code> option, then the loaded driver is automatically 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 UEFI Boot Service <code>ConnectController()</code> is called. If the UEFI driver produces the Driver Binding Protocol in the driver’s entry point, then the <code>ConnectController()</code> call exercises the <code>Supported()</code> and <code>Start()</code> services of Driver Binding Protocol that was produced.</p> <p>If the <code>-nc</code> option is used with the <code>Load</code> command, then this automatic connect operation is not performed. Instead, only the UEFI driver’s entry point is executed. When the <code>-nc</code> option is used, the UEFI Shell command <code>connect</code> can be used to connect the UEFI driver to any devices in the system that it is able to manage. The <code>Load</code> command can also take wild cards, so multiple UEFI drivers can be loaded at the same time.</p> <p>The code below shows the following examples of the <code>Load</code> command:</p> <p>Example 1: Loads and does not connect the UEFI driver image <code>EfiDriver.efi</code>. This example exercises only the UEFI driver’s entry point:</p> <pre>fs0:> Load -nc EfiDriver.efi</pre> <p>Example 2: Loads and connects the UEFI driver image called <code>EfiDriver.efi</code>. This example exercises the UEFI driver’s entry point and the <code>Supported()</code> and <code>Start()</code> functions of the Driver Binding Protocol:</p> <pre>fs0:> Load EfiDriver.efi</pre> <p>Example 3: Loads and connects all the UEFI drivers with an <code>.efi</code> extension from <code>fs0:</code>, exercising the UEFI driver entry points and their <code>Supported()</code> and <code>Start()</code> functions of the Driver Binding Protocol:</p> <pre>fs0:> Load *.efi</pre>
LoadPciRom	This command used to simulate the load of a PCI option ROM by the PCI bus driver. It also support the <code>-nc</code> flag like the <code>Load</code> command, but takes the name of a PCI Option ROM file instead of an <code>.efi</code> file.

31.3.4 Unloading UEFI drivers

This table lists UEFI Shell commands that can be used to unload a UEFI driver if it is unloadable.

Table 43—UEFI Shell commands for unloading UEFI drivers

Command	Description
Unload	<p>Unloads a UEFI driver if it is unloadable.</p> <p>Note: A UEFI driver must implement the <code>Unload()</code> service in order for the <code>Unload</code> command to be used in the UEFI Shell. If a driver does not include the <code>Unload()</code> service, the <code>Unload</code> Shell command returns unsupported.</p> <p>The <code>Unload</code> command takes a single argument that is the image handle number of the UEFI driver to unload. The <code>Dh -p Image</code> command and the <code>Drivers</code> 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 <code>Unload</code> command may fail for one of the following two reasons:</p> <ul style="list-style-type: none"> • 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. <p>Some UEFI drivers may need to be disconnected before they are unloaded. They can be disconnected with the <code>Disconnect</code> command. The following example unloads the UEFI driver on handle 27. If the UEFI driver on handle 27 is unloadable, then it has registered an <code>Unload()</code> function in its Loaded Image Protocol. This command exercises the UEFI driver's <code>Unload()</code> function:</p> <pre>Shell> Unload 27</pre>

31.3.5 Connecting UEFI Drivers

The table below lists the UEFI Shell commands that can be used to test the connecting of UEFI drivers to devices. There command support many flags, so only a few are shown in the table below.

Table 44—UEFI 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.
Disconnect	Stops UEFI drivers from managing a device.

Command	Description
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 a 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. A UEFI driver is not complete until it passes this interoperability test with the UEFI core and the full set of UEFI drivers at least 3 times in a row.

31.3.5.1 Connect

This UEFI Shell command requests UEFI drivers to start managing a device. This command tests the Driver Binding Protocol **supported()** and **start()** functions in the driver that has the specified handle. The **start()** function may create new child handles if the UEFI Driver is a bus driver or a hybrid driver.

The **Connect** command can be used to connect all UEFI drivers to all devices in the system or connect UEFI drivers to a single device. Here are several examples of using the **Connect** command:

Example 1: Connects all drivers to all devices:

```
fs0:> Connect -r
```

Example 2: Connects all drivers to the device that is abstracted by handle 23:

```
fs0:> Connect 23
```

Example 3: Connects the UEFI driver on handle 23 to the device on handle 27:

```
fs0:> Connect 23 27
```

In Example 3, note that there is a handle for the driver and a handle for the hardware device. The **Connect** command makes the connection between the two handles. The **Start()** service associates the driver with the specified hardware. If the driver needs to create a child handle for the device, it does so as part of its **start()** function. Although the handles cannot be known until the driver is executed, the handle database can be evaluated to determine the handle numbers that are passed to the connect command.

31.3.5.2 Disconnect

The **Disconnect** UEFI command stops UEFI drivers from managing a device. This command tests the Driver Binding Protocol **stop()** function in the driver.

This UEFI Shell command does not allow a driver to be disconnected unless all the child handles associated with that driver are destroyed first. Basically, this UEFI Shell command does not allow any orphans to be left in the system.

TIP: When disconnecting drivers one at a time, begin at the lowest level of child handle and work up the device tree one node at a time. The UEFI Shell command `devtree` provides a device tree view.

The code below shows the following examples of using the `Disconnect` command:

Caution: The `Disconnect` command supports a `-r` switch that can be used without any other parameters. Do NOT use this mode of the `Disconnect` command because it will disconnect all UEFI Drivers from all devices in the entire platform which typically includes the consoles devices.

Example 1: Disconnects all the UEFI drivers from the device represented by handle 23:

```
fs0:> Disconnect 23
```

Example 2: Disconnects all UEFI drivers on handle 23 and the child process (27) which was created by that driver:

```
fs0:> Disconnect 23 27
```

Example 3: Disconnects the UEFI driver represented by handle 29. The UEFI driver on handle 29 produced a child (32) and is managing a device (44), which has a device path associated with it. In order to disconnect the driver, the child and the device path managed by that driver are destroyed along with stopping the driver.

```
fs0:> Disconnect 29 32 44
```

31.3.5.3 Reconnect

The code below shows the following examples of the `Reconnect` command:

Example 1: Reconnects all the UEFI drivers to the device handle 23:

```
fs0:> Reconnect 23
```

Example 2: Reconnects the UEFI driver on handle 27 to the device on handle 23:

```
fs0:> Reconnect 23 27
```

Example 3: Reconnects all the UEFI drivers in the system:

```
fs0:> Reconnect -r
```

31.3.6 Driver and Device Information

The [following table](#) 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—UEFI Shell commands for driver and device information

Command	31.4	Description
Drivers	<p>Lists all the UEFI drivers that follow the UEFI driver model. It uses the GetDriverName() service of the Component Name protocols 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 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:</p> <p>Example 1: Shows the Drivers command being used to list the UEFI drivers in the default language.</p> <pre>fs0:> Drivers</pre> <p>Example 2: Shows the driver names in Spanish.</p> <pre>fs0:> Drivers -lsp</pre>	
Devices	<p>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 protocols 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.</p>	
DevTree	<p>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 Protocols 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:</p> <p>Example 1: Displays the device tree with the device names in the default language.</p> <pre>fs0:> DevTree</pre> <p>Example 2: Displays the device tree with the device names in Spanish.</p> <pre>fs0:> DevTree -lsp</pre> <p>Example 3: Displays the device tree with the device names shown as EFI device paths.</p> <pre>fs0:> DevTree -d</pre>	

Command	31.4	Description
Dh -d	<p>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:</p> <p>Example 1: Displays the details on the UEFI driver on handle 27.</p> <pre>fs0:> Dh -d 27</pre> <p>Example 2: Displays the details for the device on handle 23.</p> <pre>fs0:> Dh -d 23</pre> <p>Example 3: Shows details on all the drivers and devices in the system.</p> <pre>fs0:> Dh -d</pre>	
OpenInfo	Provides detailed information on a device handle 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.	

31.3.6.1 Devices

This command 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 protocols 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.

- For Component Name: use the 3-letter language localization
- For Component Name2, use the 2x3-letter language localization

The code below shows the following examples of the **Devices** command.
The **-l** switch specifies the localized language.

Example 1: Shows the **Devices** command being used to list the UEFI drivers in the default language.

```
fs0:> Devices
```

Example 2: Shows the device names in Spanish.

```
fs0:> Devices -lspa
fs0:> Devices -lsp
```

This command is backwards compatible. If the system supports both the Component Name Protocol and the Component Name2 Protocol, the driver can produce both

protocols. If the system supports only 2-letter localizations, an error is generated if an attempt is made to enter the 2-letter localization.

31.3.6.2 OpenInfo command

This command 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 may be used to display information for devices or drivers.

Example 1: 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
```

Example 2: The following example shows the **OpenInfo** command being used to display the list of devices and/or child processes being managed by a driver.

```
fs0:> OpenInfo 15
```

Example 3: The **OpenInfo** command may be used along with the **Connect**, **Disconnect**, and **Reconnect** commands to verify that a UEFI driver is opening and closing protocol interfaces correctly. For example:

```
fs0:> Connect 23  
fs0:> OpenInfo 23
```

31.3.7 Testing the Driver Configuration Protocol

The **DrvCfg** command may be used to list all devices that are being managed by UEFI drivers that support the Driver Configuration Protocols. The **Devices** and **Drivers** commands show the drivers that support the Driver Configuration Protocol and the devices that those drivers are managing or have produced. Once a device is selected, the **DrvCfg** command may be used to invoke the **SetOptions()**, **ForceDefaults()**, or **OptionsValid()** services of the Driver Configuration Protocol. The code below shows examples of using the **DrvCfg** command:

Example 1: Displays all the devices that are being managed by UEFI drivers that support the obsolete Driver Configuration Protocol.

```
fs0:> DrvCfg
```

Example 2: Forces defaults on all the devices in the system.

```
fs0:> DrvCfg -f
```

Example 3: Validates the options on all the devices in the system.

```
fs0:> DrvCfg -v
```

Example 4: Invokes the **SetOptions()** service of the Driver Configuration Protocol for the driver on handle 23 and its child process (27).

```
fs0:> DrvCfg -s 23 27
```

31.3.8 Testing the Driver Diagnostics Protocols

The DrvDiag UEFI Shell command provides the ability to test all the services of the two Driver Diagnostics Protocols that may be produced by a UEFI driver. This command is able to show the devices that are being managed by UEFI drivers that support the Driver Diagnostics Protocols. The `Devices` and `Drivers` commands show the drivers that support the Driver Diagnostics Protocols 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 Protocols. 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 Protocols.

```
fs0:> DrvDiag
```

Example 2: Invokes the `RunDiagnostics()` service of the Driver Diagnostics Protocols in standard mode for the driver on handle 15 and the device on handle 19.

```
fs0:> DrvDiag -s 15 19
```

Example 3: Invokes the `RunDiagnostics()` service of the Driver Diagnostics Protocols in manufacturing mode for the driver on handle 15 and the device on handle 19.

```
fs0:> DrvDiag -m 15 19
```

31.4 Debugging code statements

A UEFI Driver may be implemented to support both a debug (check) build and a production build. The debug build includes code that helps debug a UEFI Driver that is not included in normal production builds. UEFI Driver sources are typically implemented with all the debug build statements included. The DSC file used to build the UEFI Driver with the EDK II build tools contains statements to select a debug build or a production build with no source changes to the UEFI Driver.

The EDK II library class called `DebugLib` provides macros that 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 typically enabled only for debug builds and disabled in production builds so as to not take up any executable space. The macros available through the `DebugLib` include:

- `ASSERT (Expression)`
- `ASSERT_EFI_ERROR (Status)`
- `ASSERT_PROTOCOL_ALREADY_INSTALLED (Handle, Guid)`
- `DEBUG ((ErrorLevel, Format, . . .))`
- `DEBUG_CODE_BEGIN ()`
- `DEBUG_CODE_END ()`
- `DEBUG_CODE (Expression)`
- `DEBUG_CLEAR_MEMORY (Address, Length)`
- `CR (Record, TYPE, Field, Signature)`

These macros are described in details in the [MdePkg](#) documentation available from <http://www.tianocore.org>. The `ErrorLevel` parameter passed into the `DEBUG()` macro allows a UEFI driver to assign a different error level to each debug message, which allows debug messages to be filtered. The DSC files required to build a UEFI Driver can be used to set the `ErrorLevel` filter mask. The UEFI Shell also supports the `Err` command that allows the user to set the error level filter mask.

TIP: Use a serial port as a standard error device during debug. This a terminal emulator to be used to log debug messages to a file.

The table below contains the list of error levels that are supported in the UEFI Shell. Other levels are usable, but not defined for a specific area.

Table 46—Error levels

Mnemonic	Value	Description
<code>DEBUG_INIT</code>	0x00000001	Initialization
<code>DEBUG_WARN</code>	0x00000002	Warnings
<code>DEBUG_INFO</code>	0x00000040	Information messages
<code>DEBUG_ERROR</code>	0x80000000	Error messages.
<code>DEBUG_FS</code>	0x00000008	Used by UEFI Drivers that produce the Simple File System Protocol.
<code>DEBUG_BLKIO</code>	0x00001000	Used by UEFI Drivers that produce the Block I/O Protocols.
<code>DEBUG_NET</code>	0x00004000	Used by UEFI Drivers that produce the network protocols other than NII and UNDI.
<code>DEBUG_UNDI</code>	0x00010000	Used by UEFI Drivers that produce the NII Protocol and UNI interface.
<code>DEBUG_LOADFILE</code>	0x00020000	Used by UEFI Drivers that produce the Load File Protocol.
<code>DEBUG_EVENT</code>	0x00080000	Event messages. Used from event notification functions of UEFI Drivers.
<code>DEBUG_LOAD</code>	0x00000004	Load events. DO NOT USE.
<code>DEBUG_POOL</code>	0x00000010	Pool allocations & frees. DO NOT USE.
<code>DEBUG_PAGE</code>	0x00000020	Page allocations & frees. DO NOT USE.
<code>DEBUG_DISPATCH</code>	0x00000080	PEI/DXE/SMM Dispatchers. DO NOT USE.
<code>DEBUG_VARIABLE</code>	0x00000100	Variable. DO NOT USE.
<code>DEBUG_BM</code>	0x00000400	Boot Manager. DO NOT USE.
<code>DEBUG_GCD</code>	0x00100000	Global Coherency Database changes. DO NOT USE.
<code>DEBUG_CACHE</code>	0x00200000	Memory range cache state changes. DO NOT USE.

31.4.1 Configuring DebugLib with EDK II

The EDK II provides several methods to manage the DebugLib macros. These include:

- **MDEPKG_NDEBUG** macro
- DebugLib library instances
- DebugLib Platform Configuration Database (PCD) settings

31.4.1.1 MDEPKG_NDEBUG Define

If **MDEPKG_NDEBUG** is defined when a UEFI Driver is built, then all the **DebugLib** macros used by a UEFI Driver are removed. This provides a smaller executable, but all debug log messages, assert condition checks, and debug code are removed from the UEFI Driver that is produced by the EDK II build. The example below shows the addition of a **[BuildOptions]** section to the DSC files from [Chapter 30](#). It forces **MDEPKG_NDEBUG** to be defined for **RELEASE** builds, which means all the **DebugLib** macros are disabled when the **-b RELEASE** flag is used when building a UEFI Driver.

```
[BuildOptions]
GCC:RELEASE_*_*_CC_FLAGS = -DMDEPKG_NDEBUG
INTEL:RELEASE_*_*_CC_FLAGS = /D MDEPKG_NDEBUG
MSFT:RELEASE_*_*_CC_FLAGS = /D MDEPKG_NDEBUG
```

Example 266—EDK II Package DSC File with Build Options

31.4.1.2 DebugLib Library Instances

The **MdePkg** provides 4 different implementations of the **DebugLib** library class. These are:

- [MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf](#)
- [MdePkg/Library/BaseDebugLibConOut/BaseDebugLibConOut.inf](#)
- [MdePkg/Library/BaseDebugLibStdErr/BaseDebugLibStdErr.inf](#)
- [MdePkg/Library/BaseDebugLibSerialPort/BaseDebugLibSerialPort.inf](#)

BaseDebugLibNull is an implementation of the **DebugLib** with empty worker functions. This means the **DebugLib** macros are mapped to empty worker functions, so if the library instances is used by a UEFI Driver, no debug log messages, assert condition checks, or debug code are active. Using this library mapping is not as small as using **MDEPKG_NDEBUG**, but switching to this library mapping does not require a rebuild of the UEFI Driver sources.

BaseDebugLibStdErr is the recommended library instance for UEFI drivers that are being debugged and is the library that is used in the example DSC file in [Chapter 30](#). This sends all messages to the Standard Error console in the UEFI System Table. If there is no output, then the likely cause is that the Standard Error device is not configured. Use the platform setup to configure the Standard Error.

BaseDebugLibConOut may be used as a substitute for **BaseDebugLibStdErr** when it is not possible to get the Standard Error console configured. This sends all messages to the Standard Output console in the UEFI System Table. This mixes debug messages with the normal console activity, so the display may be difficult to read, and since most UEFI consoles do not support scroll up operations, it may be difficult to see the messages when many are displayed.

BaseDebugLibSerialPort is not a UEFI conformant DebugLib. It directly accesses serial port hardware through a **SerialPortLib** library instance. This can be useful when debugging UEFI Drivers that execute before UEFI consoles are initialized, such as UEFI Drivers that are loaded and executed from a PCI Option ROM. When this library instance is used, the UEFI Driver writer must know that there is a serial port available on the target platform under test and must configure a **SerialPortLib** with the attributes of the specific serial port that is to be used.

31.4.1.3 DebugLib Platform Configuration Database Settings

The **MdePkg** library class **DebugLib** uses several Platform Configuration Database (PCD) setting to control the behavior of the DebugLib macros. The token names for these PCD settings are as follows:

- **gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask**
- **gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel**
- **gEfiMdePkgTokenSpaceGuid.PcdDebugClearMemoryValue**

PcdDebugPropertyMask provides fine grain control over the macros provided by the **DebugLib**. The previous two sections discuss how to disable the entire **DebugLib** and how to select different **DebugLib** library instances. **PcdDebugPropertyMask** is a bit mask that allows individual **DebugLib** macro types to be enabled or disabled. The example below shows the bitmask definitions. **0x01** enables **ASSERT()** macros. **0x02** enables **DEBUG()** macros. **0x04** enables the 3 **DEBUG_CODE()** macros. **0x08** enables the **DEBUG_CLEAR_MEMORY()** macro. **0x10** and **0x20** control the behavior of the **ASSERT()** macro if the assert condition evaluates to **FALSE**. **0x10** causes a CPU breakpoint to be generated, which is useful if a source level debugger is being used, and **0x20** causes the CPU to enter an infinite loop so execution of the UEFI Driver stops.

```
//  
// Declare bits for PcdDebugPropertyMask  
//  
#define DEBUG_PROPERTY_DEBUG_ASSERT_ENABLED      0x01  
#define DEBUG_PROPERTY_DEBUG_PRINT_ENABLED       0x02  
#define DEBUG_PROPERTY_DEBUG_CODE_ENABLED        0x04  
#define DEBUG_PROPERTY_CLEAR_MEMORY_ENABLED      0x08  
#define DEBUG_PROPERTY_ASSERT_BREAKPOINT_ENABLED 0x10  
#define DEBUG_PROPERTY_ASSERT_DEADLOOP_ENABLED    0x20
```

Example 267—PcdDebugPropertyMask bitmask

PcdDebugPrintErrorLevel provides a bitmask of the debug error levels that are currently enabled. The debug print error levels are shown in the Error Levels table above. Any combination of the values can be set in the bitmask. If a bit is set, then **DEBUG()** macros with that same **ErrorLevel** bit set are printed.

`PcdDebugClearMemoryValue` provides the 8-bit byte value to use when `DEBUG_CLEAR_MEMORY()` macros are used. This value is typically set to `0x00`, but it is usually a good idea to try a few different values to make sure code is not improperly using buffer contents that have been cleared.

The following example shows the addition of a `[PcdsFixedAtBuild]` section to the DSC files from [Chapter 30](#). It sets `PcdDebugPropertyMask` so `DEBUG()`, `ASSERT()`, and `DEBUG_CODE()` macros are enabled and a breakpoint is generated when an `ASSERT()` is triggered. It also sets the `PcdDebugPrintErrorLevel` at a fairly high verbosity level with `DEBUG_ERROR`, `DEBUG_INFO`, `DEBUG_LOAD`, `DEBUG_WARN`, and `DEBUG_INIT` all enabled. Finally, it configures `PcdDebugClearMemoryValue` so `DEBUG_CLEAR_MEMORY()` macros, when they are enabled, fill buffers with `0x00`.

```
[PcdsFixedAtBuild]
gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask|0x17
gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x80000047
gEfiMdePkgTokenSpaceGuid.PcdDebugClearMemoryValue|0x00
```

Example 268—EDK II Package DSC File with Build Options

31.4.2 Capturing Debug Messages

In addition, the parameters for the serial port on the "target" system (the system under test) must be setup correctly, including baud rate, data bits, stop bits, and flow control. The settings on the target system must match the settings on the host system that is receiving the debug data. The EDK II includes sample code for serial port debug output for the PEI phase, in the MDE module package. The MDE module package also includes sample code for serial port debug output for the DXE phase

31.5 POST codes

If a 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 a UEFI driver have executed and where error conditions have been detected.

A few possibilities are presented in this discussion, but many others are possible depending on the devices that may be available on a specific platform. The first possibility is to use a POST card. Another is to use a text-mode VGA frame buffer.

It is important to note that 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 and accessing these devices may cause unexpected behavior on platforms that do not include those devices.

31.5.1 POST Card Debug

A POST card is an add-in card that displays the hex value of an 8-bit I/O write cycle to address 0x80. Some POST cards support more than 8-bits and use additional I/O port addresses such as 0x81. The EDK II MdePkg provides a library class called `PostCodeLib` that includes the `POST_CODE()` macro that may be used to abstract access to a POST

card. When a UEFI Driver is built, it can be configured in the DSC file to map the `PostCodeLib` class to the `MdePkg/Library/BasePostCodeLibPort80` instance that performs 8-bit writes to I/O port `0x80`. If a platform has the equivalent POST card functionality, but it is not located at I/O port `0x80`, an alternate implementation of the `PostCodeLib` instance can be provided that allows a UEFI Driver to send POST code values to the alternate POST card device without any source code changes to the UEFI Driver itself.

This example shows an example usage of the `POST_CODE()` macro to send POST code values of `0x10` and `0x11` as a UEFI Driver enters and leaves the driver entry point.

```
#include <Uefi.h>
#include <Library/PostCodeLib.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI
AbcDriverEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
)
{
    EFI_STATUS  Status;

    POST_CODE (0x10);

    //
    // Install driver model protocol(s) on ImageHandle
    //
    Status = EfiLibInstallDriverBinding (
        ImageHandle,           // ImageHandle
        SystemTable,           // SystemTable
        &gAbcDriverBinding,   // DriverBinding
        ImageHandle            // DriverBindingHandle
    );
    ASSERT_EFI_ERROR (Status);

    POST_CODE (0x11);

    return Status;
}
```

Example 269—UEFI Driver Entry Point with `POST_CODE()` Macros

The **PostCodeLib** uses PCDs to enable and disable the **POST_CODE()** macros. This means that **POST_CODE()** macros can be enabled during UEFI Driver development and debug when a platform with a POST card is being used, and can be easily disabled for production builds of UEFI Drivers. The example below contains a portion of the DSC file that shows how to enable **POST_CODE()** macros in a UEFI Driver.

```
[PcdsFixedAtBuild]
#
# Set POST_CODE_PROPERTY_POST_CODE_ENABLED bit (0x8) in
# PcdPostCodePropertyMask to enable POST_CODE() macros
#
gEfiMdePkgTokenSpaceGuid. PcdPostCodePropertyMask | 0x08
```

Example 270—Enable POST_CODE() macros from DSC file

31.5.2 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.

32

Distributing UEFI Drivers

Once a UEFI Driver is implemented and validated and ready to be released, there are only a few ways that the UEFI Driver can be installed onto a target platform. These include:

- PCI Option ROM on a PCI add-in card.
- Integrated into the platform firmware FLASH image.
- A file in an EFI System Partition.

32.1 PCI Option ROM

[Chapter 18](#) covers the guidelines for implementing a UEFI Driver for a PCI device. The end of that chapter covers how to use the tools provided with the EDK II to convert one or more UEFI Drivers in PE/COFF image formats into a single PCI Option ROM image that may be included with a PCI add-in card.

When a PCI add-in card is installed into a target platform, the PCI Option ROM contents are discovered by the PCI Bus Driver and UEFI Drivers are loaded and executed automatically. No additional platform configuration should be required. Some platforms may provide setup options to enable/disable specific PCI slots or enable/disable the loading of PCI Option ROMs. If a UEFI Driver stored in a PCI Option ROM is not being loaded and executed, then check the platform firmware configuration screens.

32.2 Integrated in Platform FLASH

A manufacturer that produces UEFI Drivers for their devices may choose to work with platform manufacturers to have their UEFI Drivers integrated into the UEFI firmware for a target platform. UEFI Drivers can be provided in source or binary form, and it is up to the platform manufacturer to integrate those UEFI Drivers into their UEFI platform firmware images and verify that the UEFI Driver is loaded and executed properly.

32.3 EFI System Partition

A device manufacturer that produces UEFI Drivers for their devices may choose to have their UEFI Drivers installed onto an EFI System Partition on a target platform. This method may be selected for UEFI Drivers that cannot be distributed using the two methods described above. It may also be a convenient method for UEFI Driver updates to be distributed and installed onto target platforms. See the *UEFI Specification* for details on EFI System Partitions and <http://www.uefi.org> for details on how device manufacturers can reserve a subdirectory name for use on EFI System Partitions.

Note: *There is no requirement for UEFI conformant platform firmware implementations to load UEFI Drivers from EFI System Partitions. The capability is defined by the UEFI Specification but there is no requirement that all platforms implement this capability.*

Since this method depends on being able to access the EFI System Partition, the UEFI Driver that is to be installed on the EFI System Partition must not be required to access the EFI System Partition itself, either directly or indirectly. For example, a UEFI Driver for a SCSI Host Controller cannot be installed on an EFI System Partition of a mass storage device attached to that same SCSI Host Controller. However, if the SCSI Host Controller is a PCI add-in card with a UEFI Driver in a PCI Option ROM or the UEFI Driver for the SCSI Host Controller is integrated in the platform firmware, it may be possible to install an update to the UEFI Driver for the SCSI Host Controller in an EFI System Partition on one of the mass storage devices attached to that SCSI Host Controller as long as the UEFI Driver in the PCI Option ROM or the platform firmware is functional enough to load the updated UEFI Driver from the EFI System Partition.

There are a few steps that must be performed in order for a UEFI Driver to be installed onto an EFI System Partition and for that UEFI Driver to be automatically loaded and executed each time the target platform is booted.

- 1) The UEFI Driver must be copied onto a mass storage device that contains an EFI System Partition. This may require a custom UEFI Application to perform this transfer, or utilities such as 1 the UEFI Shell and UEFI Shell scripts may be used to install a UEFI Driver into a device specific directory.
- 2) Update the `Driver####` and `DriverOrder` UEFI variables so the UEFI Driver installed on the EFI System Partition is automatically loaded and executed on every boot. These variables can be updated from a custom UEFI Application, or OEM setup screens if this option is exposed.

Tip: Use the UEFI Shell `drivers` command to view the set of UEFI Drivers that have been loaded and executed to verify that a UEFI Driver that has been installed and configured to load from EFI System Partition has actually been loaded and executed by the platform firmware.

Appendix A

EDK II File Templates

This discussion contains templates and guidelines for creating files for protocols, GUIDs, EDK II Library Classes, and UEFI drivers in EDK II packages. The naming conventions for the driver entry point and the functions exported by a driver that are presented here guarantee that a unique name is produced for every function, which aides in call stack analysis when root-causing driver issues. The Doxygen style function header comment blocks have been removed the file templates shown in this appendix to highlight the source code elements required to build a UEFI Driver. The function headers comments blocks can be added by coping them from the EDK II **MdePkg** protocol include files located in the **MdePkg/Include/Protocol/** directory.

The following expressions are used throughout this chapter to show where protocol names, GUID names, function names, and driver names should be substituted in a file template:

<<PackageName>>

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

<<BriefDescription>>

One line brief description of a file or library or module.

<<DetailedDescription>>

Paragraph that is a detailed description of a file or library or module.

<<Copyright>>

One or more copyright declarations for a file or library or module.

<<License>>

One or more licenses for a file or library or module.

<<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**).

<<GUID_STRUCT>>

Represents the GUID value in the format of a C data structure. New GUID values can be generated using the **GUIDGEN** utility shipped with Microsoft™ compilers, or the **uuidgen** command under Linux. (e.g., { 0x9e34954, 0x6c5, 0x4e1a, { 0xb7, 0xeb, 0x5d, 0x5c, 0x9, 0xca, 0x6d, 0xaf } })

<<GUID_REGISTRY_FORMAT>>

Represents the GUID value in Registry Format. New GUID values can be generated

using the **GUIDGEN** utility shipped with Microsoft® compilers, or the **uuidgen** command under Linux. (e.g., **1E9CD853-7A32-49e0-8140-145CD35C6632**)

<<DriverVersion>>

A 32-bit value representation for the version of the UEFI Driver used to fill in the **Version** field of the Driver Binding Protocol. (e.g., **0x00000010**).

<<DriverVersionString>>

A text string representation for the version of the UEFI Driver. (e.g., 1.7).

<<FunctionNameN>>

Represents the *n*th 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_NAME>>

Represents the *n*th 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 0xFFFFFFFF0–0xFFFFFFFF are reserved for UEFI drivers that are written by OEMs for integrated devices. Values from 0x10–0xFFFFFFFf are reserved for UEFI drivers that are written by IHVs.

<<Iso639SupportedLanguages>>

A null terminated ASCII string of one or more 3 character ISO 639-2 language code with no separator character. (e.g. "eng" for English, "engfra" for English and French).

<<Rfc4646SupportedLanguages>>

A null terminated ASCII string of one or more RFC 4646 language codes separated by semicolons ';' (e.g. "en" for English, "en-US;fr" for U.S. English and French).

<<UEFI_SYSTEM_TABLE_REVISION>>

The 32-bit revision of the *UEFI Specification* that the UEFI Driver requires to run

correctly. Usually, one of the define names from `<Uefi.h>` is used, which includes the following:

```
EFI_2_31_SYSTEM_TABLE_REVISION
EFI_2_30_SYSTEM_TABLE_REVISION
EFI_2_20_SYSTEM_TABLE_REVISION
EFI_2_10_SYSTEM_TABLE_REVISION
EFI_2_00_SYSTEM_TABLE_REVISION
EFI_1_10_SYSTEM_TABLE_REVISION
EFI_1_02_SYSTEM_TABLE_REVISION
```

<<ProtocolNameCn>>

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

<<PROTOCOL_NAME_PM>>

Represents the *m*th name of a protocol that is produced by a 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`).

<<UsbSpecificationMajorRevision>>

Denotes the major revision of the *USB Specification* that a USB host controller driver follows (e.g. 1 for the *USB 1.1 Specification*).

<<UsbSpecificationMinorRevision>>

Denotes the minor revision of that *USB Specification* that a USB host controller driver follows (e.g. 0 for the *USB 2.0 Specification*).

A.1

UEFI Driver Template

UEFI driver sources are typically placed in a EDK II package. There are no restrictions on the directory structure within an EDK II package. A common convention for UEFI Drivers related to industry standard busses is to place the UEFI Driver in a directory path the such as `<<PackageName>>/Bus/<<BUSTYPE>>/<<DriverName>>`. The directory structure for a single UEFI Driver does not have to be flat. Multiple closely related UEFI Drivers may be placed in subdirectories. The directory name for a UEFI driver is typically of the form `<<DriverName>>`. For example, the USB keyboard driver in the `MdeModulePkg` is located in the directory `\MdeModulePkg\Bus\Usb\UsbKb`.

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

- `<<DriverName>>.inf`
- `<<DriverName>>.h`
- `<<DriverName>>.c`

It is possible to reduce the number of files down to just `<>DriverName>.inf` and `<>DriverName>.c`. However, if the complexity of the UEFI Driver increases over time where a splitting out a second .c file makes sense, then a common .h file is usually required. If a UEFI Driver is implemented with a common .h file from the beginning, then additional .c file can be added without have to reorganize the other source files.

The `<>DriverName>.inf` file describes the information the EDK II build system required to build UEFI Driver into a UEFI conformant executable image. This includes elements such as source filenames, EDK II package dependencies, libraries that are used, Protocols that are produced/consumed, and GUIDs that are used.

The `<>DriverName>.h` file includes the standard UEFI include file, include files for libraries that the UEFI Driver uses, and include files for protocols or GUIDs that the UEFI Driver either produces or consumes. In addition, the `<>DriverName>.h` file may contain the function prototypes for the public APIs that are produced by the UEFI Driver and declarations for `#defines` and data structures that are internal to the UEFI Driver implementation.

The `<>DriverName>.c` file contains the driver entry point. If a 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 other 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, the Component Name Protocol, and the Component Name2 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 and the Component Name2 Protocol.

A.1.1

<>DriverName>.inf File for a UEFI Driver

A UEFI Driver module information file typically consists of the following elements. The [following example](#) shows a template of an INF file with these same elements.

- [Defines] section that declares a name for GUID for the UEFI Driver along with the name of the function that is the entry point to the UEFI Driver.
- [Sources] section with the list of .c and .h files required to build the UEFI Driver.
- [Packages] section with the list of EDK II packages that the UEFI Driver requires to build. All UEFI Drivers use MdePkg/MdePkg.dec for the definitions from the *UEFI Specification*. If a UEFI Driver implementation uses Protocols or GUIDs declared in other EDK II Packages, then those packages must be listed in this section too.
- [LibraryClasses] section with the list of libraries that the UEFI Driver uses.
- [Protocols] section with the list of protocols that the UEFI Driver produces or consumes.
- [Guids] section with the list of GUIDs that the UEFI Driver produces or consumes.

```

## @file
# <<BriefDescription>>
#
# <<DetailedDescription>>
#
# <<Copyright>>
#
# <<License>>
#
##

[Defines]
INF_VERSION = 0x00010005
BASE_NAME = <<DriverName>>
FILE_GUID = <<GUID_REGISTRY_FORMAT>>
MODULE_TYPE = UEFI_DRIVER
VERSION_STRING = <<DriverVersionString>>
ENTRY_POINT = <<DriverName>>DriverEntryPoint

[Sources]
<<DriverName>>.h
<<DriverName>>.c

[Packages]
MdePkg/MdePkg.dec
#
# List other packages that the UEFI Driver depends upon
#
<<PackageName>>/<<PackageName>>.dec

[LibraryClasses]
UefiDriverEntryPoint
UefiBootServicesTableLib
MemoryAllocationLib
BaseMemoryLib
BaseLib
UefiLib
DevicePathLib
DebugLib
#
# List of additional libraries that the UEFI Driver uses
#
[Protocols]
#
# List of Protocols the UEFI Driver produces or consumes
#
gEfi<<ProtocolName>>ProtocolGuid

[Guids]
#
# List of GUIDs the UEFI Driver produces or consumes
#
gEfi<<GuidName>>Guid

```

Example A-1—UEFI Driver INF file template

A.1.2

<<DriverName>>.inf File for a UEFI Runtime Driver

The requirements for the module information file for a UEFI Runtime Driver are slightly different than UEFI Drivers. The **MODULE_TYPE** must be set to **DXE_RUNTIME_DRIVER** and the INF file must include a fixed **[Depex]** section. All other requirements are the same. The

example below shows a template of an INF file for a UEFI Runtime Driver and also adds the `UefiRuntimeServicesTableLib` and `UefiRuntimeLib` to the `[LibraryClasses]` section because those two library classes are commonly used by UEFI Runtime Drivers.

```
## @file
# <<BriefDescription>>
#
# <<DetailedDescription>>
#
# <<Copyright>>
#
# <<License>>
#
##

[Defines]
INF_VERSION = 0x00010005
BASE_NAME = <<DriverName>>
FILE_GUID = <<GUID_REGISTRY_FORMAT>>
MODULE_TYPE = DXE_RUNTIME_DRIVER
VERSION_STRING = <<DriverVersionString>>
ENTRY_POINT = <<DriverName>>DriverEntryPoint

[Sources]
<<DriverName>>.h
<<DriverName>>.c

[Packages]
MdePkg/MdePkg.dec
#
# List other packages that the UEFI Driver depends upon
#
<<PackageName>>/<<PackageName>>.dec

[LibraryClasses]
UefiDriverEntryPoint
UefiBootServicesTableLib
MemoryAllocationLib
BaseMemoryLib
BaseLib
UefiLib
DevicePathLib
DebugLib
UefiRuntimeServicesTableLib
UefiRuntimeLib
#
# List of additional libraries that the UEFI Driver uses
#
[EfiProtocol]
#
# List of Protocols the UEFI Driver produces or consumes
#
gEfi<<ProtocolName>>ProtocolGuid

[Guids]
#
# List of GUIDs the UEFI Driver produces or consumes
#
gEfi<<GuidName>>Guid

[Depex]
gEfiBdsArchProtocolGuid AND
gEfiCpuArchProtocolGuid AND
gEfiMetronomeArchProtocolGuid AND
```

```

gEfiMonotonicCounterArchProtocolGuid AND
gEfiRealTimeClockArchProtocolGuid AND
gEfiResetArchProtocolGuid AND
gEfiRuntimeArchProtocolGuid AND
gEfiSecurityArchProtocolGuid AND
gEfiTimerArchProtocolGuid AND
gEfiVariableWriteArchProtocolGuid AND
gEfiVariableArchProtocolGuid AND
gEfiWatchdogTimerArchProtocolGuid

```

Example A-2—UEFI Runtime Driver INF file template

A.1.3

<<DriverName>>.h File

A 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

This example shows a template for a UEFI Driver include file.

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

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

#include <Uefi.h>

//
// Include Protocols that are consumed
//

```

```

#include <Protocol/<<ProtocolNameC1>>.h>
#include <Protocol/<<ProtocolNameC2>>.h>
// .
#include <Protocol/<<ProtocolNameCn>>.h>

//
// Include Protocols that are produced
//
#include <Protocol/<<ProtocolNameP1>>.h>
#include <Protocol/<<ProtocolNameP2>>.h>
// .
#include <Protocol/<<ProtocolNamePm>>.h>

//
// Include GUIDs that are consumed
//
#include <Guid/<<GuidName1>>.h>
#include <Guid/<<GuidName2>>.h>
// .
#include <Guid/<<GuidNamep>>.h>

//
// Include Library Classes commonly used by UEFI Drivers
//
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>
#include <Library/BaseMemoryLib.h>
#include <Library/BaseLib.h>
#include <Library/UefiLib.h>
#include <Library/DevicePathLib.h>
#include <Library/DebugLib.h>

//
// Include additional Library Classes that are used
//
#include <Library/<<LibraryName1>>.h>
#include <Library/<<LibraryName2>>.h>
// .
#include <Library/<<LibraryNameq>>.h>

//
// Define driver version Driver Binding Protocol
//
#define <<DRIVER_NAME>>_VERSION <<DriverVersion>>

//
// Private Context Data Structure
//
#define <<DRIVER_NAME>>_PRIVATE_DATA_SIGNATURE 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 depending on driver features
//
extern EFI_COMPONENT_NAME2_PROTOCOL      g<<DriverName>>ComponentName2;
extern EFI_HII_CONFIG_ACCESS_PROTOCOL    g<<DriverName>>ConfigAccess;
extern EFI_DRIVER_DIAGNOSTICS2_PROTOCOL  g<<DriverName>>DriverDiagnostics2;
extern EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL g<<DriverName>>DriverFamilyOverride;
extern EFI_DRIVER_HEALTH_PROTOCOL        g<<DriverName>>DriverHealth;

//
// Optional Global Variables for compatibility with UEFI 2.0
//
extern EFI_DRIVER_CONFIGURATION2_PROTOCOL g<<DriverName>>DriverConfiguration2;

//
// Optional Global Variables for compatibility with EFI 1.10
//
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 prototypes for the APIs in the Produced Protocols
//
#endif

```

Example A-3—UEFI Driver include file template

A.1.4

<<DriverName>>.c File

A UEFI source file contains:

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

The following example shows a template for the main source file of a UEFI Driver that follows the UEFI Driver Model and produces the Driver Binding Protocol. The structure for the Driver Supported EFI Version Protocol is also declared, but is not installed in the Driver Entry Point because that protocol is optional. This template contains a template of an empty function from additional protocols that the UEFI Driver may produce. The functions from the various protocols that a UEFI driver may produce are discussed in later sections. There are many optional UEFI Driver features that are not shown in this specific template. Each of those optional features are discussed in earlier chapters along with details on how to add each of those optional features to a UEFI Driver.

```
/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL
g<<DriverName>>DriverSupportedEfiVersion = {
    sizeof (EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL),
    <<UEFI_SYSTEM_TABLE_REVISION>>
};

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

EFI_STATUS
EFIAPI
<<DriverName>>DriverEntryPoint (
    IN EFI_HANDLE     ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    //
    // Install UEFI Driver Model protocol(s).
    //
    Status = EfiLibInstallDriverBindingComponentName2 (
        ImageHandle,
        SystemTable,
        &g<<DriverName>>DriverBinding,
        ImageHandle,
        &g<<DriverName>>ComponentName,
    );
}
```

```

        &g<<DriverName>>ComponentName2
    );
ASSERT_EFI_ERROR (Status);

return Status;
}

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

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

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

//
// 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 A-4—UEFI Driver implementation template

A.1.5

<<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.

A UEFI Driver protocol source file contains:

- `#include` statement for `<<DriverName>>.h`.
- Global variable declaration. This declaration applies only to protocols such as the Component Name Protocols 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](#) of this guide).
- Implementation of the APIs from the produced protocols.

The template in the example below shows the main source file for a protocol produced by a UEFI driver. This template contains empty protocol function implementations. The remaining sections of this appendix shows template files for all the optional UEFI Driver protocols.

```
/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#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
// parameter.
//
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 A-5—UEFI Driver protocol implementation template

A.2

UEFI Driver Optional Protocol Templates

This section contains templates for the implementation of optional protocols that may be part of a UEFI Driver implementation. This includes the following:

- Component Name Protocols
- Driver Configuration Protocols
- HII Config Access Protocol
- Driver Health Protocol
- Driver Family Override Protocol
- Bus Specific Driver Override Protocol
- Driver Diagnostics Protocols

A.2.1

ComponentName.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME_PROTOCOL g<<DriverName>>ComponentName = {
    (EFI_COMPONENT_NAME_GET_DRIVER_NAME)    <<DriverName>>ComponentNameGetDriverName,
    (EFI_COMPONENT_NAME_GET_CONTROLLER_NAME)<<DriverName>>ComponentNameGetControllerName,
    "<<Iso639SupportedLanguages>>"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_COMPONENT_NAME2_PROTOCOL g<<DriverName>>ComponentName2 = {
    <<DriverName>>ComponentNameGetDriverName,
    <<DriverName>>ComponentNameGetControllerName,
    "<<Rfc4646SupportedLanguages>>"
};

EFI_STATUS
EFIAPI
<<DriverName>>ComponentNameGetDriverName (
    IN EFI_COMPONENT_NAME2_PROTOCOL *This,
    IN CHAR8                      *Language,
    OUT CHAR16                     **DriverName
)

```

```

{
}

EFI_STATUS
EFIAPI
<<DriverName>>ComponentNameGetControllerName (
    IN  EFI_COMPONENT_NAME2_PROTOCOL   *This,
    IN  EFI_HANDLE                   ControllerHandle,
    IN  EFI_HANDLE                   ChildHandle     OPTIONAL,
    IN  CHAR8                        *Language,
    OUT CHAR16                       **ControllerName
)
{
}

```

Example A-6—Component Name Protocol implementation template

A.2.2 *DriverConfiguration.c File*

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_CONFIGURATION_PROTOCOL g<<DriverName>>DriverConfiguration = {
    (EFI_DRIVER_CONFIGURATION_SET_OPTIONS) <<DriverName>>DriverConfigurationSetOptions,
    (EFI_DRIVER_CONFIGURATION_OPTIONS_VALID) <<DriverName>>DriverConfigurationOptionsValid,
    (EFI_DRIVER_CONFIGURATION_FORCE_DEFAULTS)<<DriverName>>DriverConfigurationForceDefaults,
    "<<Iso639SupportedLanguages>>"
};

///
/// Driver Configuration 2 Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_CONFIGURATION2_PROTOCOL g<<DriverName>>DriverConfiguration2 = {
    <<DriverName>>DriverConfigurationSetOptions,
    <<DriverName>>DriverConfigurationOptionsValid,
    <<DriverName>>DriverConfigurationForceDefaults,
    "<<Rfc4646SupportedLanguages>>"
};

EFI_STATUS
EFIAPI
<<DriverName>>DriverConfigurationSetOptions (
    IN  EFI_DRIVER_CONFIGURATION2_PROTOCOL           *This,
    IN  EFI_HANDLE                    ControllerHandle,
    IN  EFI_HANDLE                    ChildHandle     OPTIONAL,
    IN  CHAR8                         *Language,
    OUT EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED *ActionRequired
)
{
    return EFI_UNSUPPORTED;
}

```

```

}

EFI_STATUS
EFIAPI
<<DriverName>>DriverConfigurationOptionsValid (
    IN EFI_DRIVER_CONFIGURATION2_PROTOCOL           *This,
    IN EFI_HANDLE                                     ControllerHandle,
    IN EFI_HANDLE                                     ChildHandle OPTIONAL
)
{
    return EFI_UNSUPPORTED;
}

EFI_STATUS
EFIAPI
<<DriverName>>DriverConfigurationForceDefaults (
    IN EFI_DRIVER_CONFIGURATION2_PROTOCOL           *This,
    IN EFI_HANDLE                                     ControllerHandle,
    IN EFI_HANDLE                                     ChildHandle OPTIONAL,
    IN UINT32                                         DefaultType,
    OUT EFI_DRIVER_CONFIGURATION_ACTION_REQUIRED    *ActionRequired
)
{
    return EFI_UNSUPPORTED;
}

```

Example A-7—Driver Configuration Protocol implementation template

A.2.3

HiiConfigAccess.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_HII_CONFIG_ACCESS_PROTOCOL g<<DriverName>>HiiConfigAccess = {
    <<DriverName>>HiiConfigAccessExtractConfig,
    <<DriverName>>HiiConfigAccessRouteConfig,
    <<DriverName>>HiiConfigAccessCallback
};

EFI_STATUS
EFIAPI
<<DriverName>>HiiConfigAccessExtractConfig (
    IN CONST  EFI_HII_CONFIG_ACCESS_PROTOCOL  *This,
    IN CONST  EFI_STRING                      Request,
    OUT      EFI_STRING                      *Progress,
    OUT      EFI_STRING                      *Results
)

```

```

{
}

EFI_STATUS
EFIAPI
<<DriverName>>HiiConfigAccessRouteConfig (
    IN CONST  EFI_HII_CONFIG_ACCESS_PROTOCOL *This,
    IN CONST  EFI_STRING                    Configuration,
    OUT       EFI_STRING                   *Progress
)
{
}
}

EFI_STATUS
EFIAPI
<<DriverName>>HiiConfigAccessCallback (
    IN      CONST EFI_HII_CONFIG_ACCESS_PROTOCOL *This,
    IN      EFI_BROWSER_ACTION                 Action,
    IN      EFI_QUESTION_ID                  QuestionId,
    IN      UINT8                           Type,
    IN OUT  EFI_IFR_TYPE_VALUE              *Value,
    OUT     EFI_BROWSER_ACTION_REQUEST     *ActionRequest
)
{
}

```

Example A-8—Driver Health Protocol implementation template

A.2.4

DriverHealth.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_HEALTH_PROTOCOL g<<DriverName>>DriverHealth = {
    <<DriverName>>DriverHealthGetHealthStatus,
    <<DriverName>>DriverHealthRepair
};

EFI_STATUS
EFIAPI
<<DriverName>>DriverHealthGetHealthStatus (
    IN  EFI_DRIVER_HEALTH_PROTOCOL      *This,
    IN  EFI_HANDLE                      ControllerHandle OPTIONAL,
    IN  EFI_HANDLE                      ChildHandle   OPTIONAL,
    OUT EFI_DRIVER_HEALTH_STATUS       *HealthStatus,
    OUT EFI_DRIVER_HEALTH_HII_MESSAGE **MessageList  OPTIONAL,
    OUT EFI_HII_HANDLE                 *FormHiiHandle OPTIONAL
)
{
    return EFI_UNSUPPORTED;
}

```

```

EFI_STATUS
EFIAPI
<<DriverName>>DriverHealthRepair (
    IN  EFI_DRIVER_HEALTH_PROTOCOL           *This,
    IN  EFI_HANDLE                          ControllerHandle,
    IN  EFI_HANDLE                          ChildHandle      OPTIONAL,
    IN  EFI_DRIVER_HEALTH_REPAIR_PROGRESS_NOTIFY ProgressNotification OPTIONAL
)
{
    return EFI_UNSUPPORTED;
}

```

Example A-9—Driver Health Protocol implementation template

A.2.5

DriverFamilyOverride.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL g<<DriverName>>DriverFamilyOverride =
{
    <<DriverName>>DriverFamilyOverrideGetVersion
};

UINT32
EFIAPI
<<DriverName>>DriverFamilyOverrideGetVersion (
    IN EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL  *This
)
{
    return 0;
}

```

Example A-10—Driver Family Override Protocol implementation template

A.2.6

BusSpecificDriverOverride.c File

```
/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL
g<<DriverName>>BusSpecificDriverOverride = {
    <<DriverName>>BusSpecificDriverOverrideGetDriver
};

EFI_STATUS
EFIAPI
<<DriverName>>BusSpecificDriverOverrideGetDriver (
    IN EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL *This,
    IN OUT EFI_HANDLE                           *DriverImageHandle
)
{
    return EFI_NOT_FOUND;
}
```

Example A-11—Bus Specific Driver Override Protocol implementation template

A.2.7

DriverDiagnostics.c File

```
/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_DIAGNOSTICS_PROTOCOL g<<DriverName>>DriverDiagnostics = {
    (EFI_DRIVER_DIAGNOSTICS_RUN_DIAGNOSTICS)<<DriverName>>DriverDiagnosticsRunDiagnostics,
    "<<Iso639SupportedLanguages>>"
};

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_DRIVER_DIAGNOSTICS2_PROTOCOL g<<DriverName>>DriverDiagnostics2 = {
    <<DriverName>>DriverDiagnosticsRunDiagnostics,
```

```

" <<Rfc4646SupportedLanguages>> "
};

EFI_STATUS
EFIAPI
<<DriverName>>DriverDiagnosticsRunDiagnostics (
    IN EFI_DRIVER_DIAGNOSTICS2_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    IN EFI_HANDLE ChildHandle OPTIONAL,
    IN EFI_DRIVER_DIAGNOSTIC_TYPE DiagnosticType,
    IN CHAR8 *Language,
    **ErrorType,
    *BufferSize,
    **Buffer
)
{
}

```

Example A-12—Driver Diagnostics Protocols implementation template

A.3 *UEFI Driver I/O Protocol Templates*

This section contains templates for the implementation of protocols that provide I/O services or services to abstract a specific type of device hardware. This includes the following:

- USB Host Controllers
- SCSI Host Controllers
- ATA Host Controllers
- Simple Text Input Devices
- Simple Text Output Devices
- Serial Port (UART) Controllers
- Graphics Controllers
- Network Interface Controllers
- Mass Storage Device (Hard Disk, CD-ROM, DVD-ROM, FLASH drive)
- User Credential Devices (Smart Card, Fingerprint Readers, etc.)

A.3.1 *Usb2Hc.c File*

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

```

```

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_USB2_HC_PROTOCOL g<<DriverName>>Usb2HostController = {
    <<DriverName>>Usb2HostControllerGetCapability,
    <<DriverName>>Usb2HostControllerReset,
    <<DriverName>>Usb2HostControllerGetState,
    <<DriverName>>Usb2HostControllerSetState,
    <<DriverName>>Usb2HostControllerControlTransfer,
    <<DriverName>>Usb2HostControllerBulkTransfer,
    <<DriverName>>Usb2HostControllerAsyncInterruptTransfer,
    <<DriverName>>Usb2HostControllerSyncInterruptTransfer,
    <<DriverName>>Usb2HostControllerIsochronousTransfer,
    <<DriverName>>Usb2HostControllerAsyncIsochronousTransfer,
    <<DriverName>>Usb2HostControllerGetRootHubPortStatus,
    <<DriverName>>Usb2HostControllerSetRootHubPortFeature,
    <<DriverName>>Usb2HostControllerClearRootHubPortFeature,
    <<UsbSpecificationMajorRevision>>,
    <<UsbSpecificationMinorRevision>>
};

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerGetCapability (
    IN EFI_USB2_HC_PROTOCOL *This,
    OUT UINT8              *MaxSpeed,
    OUT UINT8              *PortNumber,
    OUT UINT8              *Is64BitCapable
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerReset (
    IN EFI_USB2_HC_PROTOCOL *This,
    IN UINT16               Attributes
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerGetState (
    IN       EFI_USB2_HC_PROTOCOL   *This,
    OUT      EFI_USB_HC_STATE     *State
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerSetState (
    IN EFI_USB2_HC_PROTOCOL   *This,
    IN EFI_USB_HC_STATE       State
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerControlTransfer (
    IN       EFI_USB2_HC_PROTOCOL           *This,
    IN       UINT8                         DeviceAddress,
    IN       UINT8                         DeviceSpeed,
    IN       UINTN                         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_USB2_HC_TRANSACTION_TRANSLATOR     *Translator,
OUT     UINT32                                     *TransferResult
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerBulkTransfer (
    IN      EFI_USB2_HC_PROTOCOL                *This,
    IN      UINT8                                DeviceAddress,
    IN      UINT8                                EndPointAddress,
    IN      UINT8                                DeviceSpeed,
    IN      UINTN                                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_USB2_HC_TRANSACTION_TRANSLATOR  *Translator,
    OUT     UINT32                                *TransferResult
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerAsyncInterruptTransfer (
    IN      EFI_USB2_HC_PROTOCOL                *This,
    IN      UINT8                                DeviceAddress,
    IN      UINT8                                EndPointAddress,
    IN      UINT8                                DeviceSpeed,
    IN      UINTN                                MaximumPacketLength,
    IN      BOOLEAN                               IsNewTransfer,
    IN OUT UINT8                                *DataToggle,
    IN      UINTN                                PollingInterval OPTIONAL,
    IN      UINTN                                DataLength OPTIONAL,
    IN      EFI_USB2_HC_TRANSACTION_TRANSLATOR  *Translator OPTIONAL,
    IN      EFI_ASYNC_USB_TRANSFER_CALLBACK     CallBackFunction OPTIONAL,
    IN      VOID                                 *Context OPTIONAL
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerSyncInterruptTransfer (
    IN      EFI_USB2_HC_PROTOCOL                *This,
    IN      UINT8                                DeviceAddress,
    IN      UINT8                                EndPointAddress,
    IN      UINT8                                DeviceSpeed,
    IN      UINTN                                MaximumPacketLength,
    IN OUT VOID                                 *Data,
    IN OUT UINTN                                *DataLength,
    IN OUT UINT8                                *DataToggle,
    IN      UINTN                                TimeOut,
    IN      EFI_USB2_HC_TRANSACTION_TRANSLATOR  *Translator,
    OUT     UINT32                                *TransferResult
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerIsochronousTransfer (

```

```

IN      EFI_USB2_HC_PROTOCOL           *This,
IN      UINT8                          DeviceAddress,
IN      UINT8                          EndPointAddress,
IN      UINT8                          DeviceSpeed,
IN      UINTN                          MaximumPacketLength,
IN      UINT8                          DataBuffersNumber,
IN      VOID                           *Data[EFI_USB_MAX_ISO_BUFFER_NUM],
IN      UINTN                          DataLength,
IN      EFI_USB2_HC_TRANSACTION_TRANSLATOR *Translator,
OUT     UINT32                         *TransferResult
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerAsyncIsochronousTransfer (
    IN      EFI_USB2_HC_PROTOCOL           *This,
    IN      UINT8                          DeviceAddress,
    IN      UINT8                          EndPointAddress,
    IN      UINT8                          DeviceSpeed,
    IN      UINTN                          MaximumPacketLength,
    IN      UINT8                          DataBuffersNumber,
    IN      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
EFIAPI
<<DriverName>>Usb2HostControllerGetRootHubPortStatus (
    IN      EFI_USB2_HC_PROTOCOL           *This,
    IN      UINT8                          PortNumber,
    OUT    EFI_USB_PORT_STATUS           *PortStatus
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerSetRootHubPortFeature (
    IN EFI_USB2_HC_PROTOCOL           *This,
    IN UINT8                          PortNumber,
    IN EFI_USB_PORT_FEATURE          PortFeature
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>Usb2HostControllerClearRootHubPortFeature (
    IN EFI_USB2_HC_PROTOCOL           *This,
    IN UINT8                          PortNumber,
    IN EFI_USB_PORT_FEATURE          PortFeature
)
{
}

```

Example A-13—USB 2 Host Controller Protocol implementation template

A.3.2

ExtScsiPassThru.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_EXT_SCSI_PASS_THRU_PROTOCOL g<<DriverName>>ExtScsiPassThru = {
    NULL,
    <<DriverName>>ExtScsiPassThruPassThru,
    <<DriverName>>ExtScsiPassThruGetNextTargetLun,
    <<DriverName>>ExtScsiPassThruBuildDevicePath,
    <<DriverName>>ExtScsiPassThruGetTargetLun,
    <<DriverName>>ExtScsiPassThruResetChannel,
    <<DriverName>>ExtScsiPassThruResetTargetLun,
    <<DriverName>>ExtScsiPassThruGetNextTarget
};

EFI_STATUS
EFIAPI
<<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
EFIAPI
<<DriverName>>ExtScsiPassThruGetNextTargetLun (
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL           *This,
    IN OUT UINT8                                 **Target,
    IN OUT UINT64                                *Lun
)
{
}

EFI_STATUS
EFIAPI
<<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
EFIAPI
<<DriverName>>ExtScsiPassThruGetTargetLun (
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL           *This,
    IN EFI_DEVICE_PATH_PROTOCOL                  *DevicePath,
    OUT UINT8                                     **Target,
    OUT UINT64                                     *Lun
)
{
}

```

```

        )
{
}

EFI_STATUS
EFIAPI
<<DriverName>>ExtScsiPassThruResetChannel (
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL           *This
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>ExtScsiPassThruResetTargetLun (
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL           *This,
    IN UINT8                                      *Target,
    IN UINT64                                     Lun
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>ExtScsiPassThruGetNextTarget (
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL           *This,
    IN OUT UINT8                                 **Target
)
{
}

```

Example A-14—Extended SCSI Pass Thru Protocol implementation template

A.3.3

AtaPassThru.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

GLOBAL_REMOVE_IF_UNREFERENCED
EFI_ATA_PASS_THRU_PROTOCOL  g<<DriverName>>AtaScsiPassThru =  {
    NULL,
    <<DriverName>>AtaPassThruPassThru,
    <<DriverName>>AtaPassThruGetNextPort,
    <<DriverName>>AtaPassThruGetNextDevice,
    <<DriverName>>AtaPassThruBuildDevicePath,
    <<DriverName>>AtaPassThruGetDevice,
    <<DriverName>>AtaPassThruResetPort,
    <<DriverName>>AtaPassThruResetDevice
};

EFI_STATUS
EFIAPI
<<DriverName>>AtaPassThruPassThru (
    IN     EFI_ATA_PASS_THRU_PROTOCOL           *This,
    IN     UINT16                                Port,
    IN     UINT16                                PortMultiplierPort,

```

```

IN OUT EFI_ATA_PASS_THRU_COMMAND_PACKET *Packet,
IN      EFI_EVENT                      Event OPTIONAL
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>AtaPassThruGetNextPort (
    IN EFI_ATA_PASS_THRU_PROTOCOL *This,
    IN OUT UINT16                *Port
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>AtaPassThruGetNextDevice (
    IN EFI_ATA_PASS_THRU_PROTOCOL *This,
    IN UINT16                     Port,
    IN OUT UINT16                *PortMultiplierPort
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>AtaPassThruBuildDevicePath (
    IN      EFI_ATA_PASS_THRU_PROTOCOL *This,
    IN      UINT16                   Port,
    IN      UINT16                   PortMultiplierPort,
    IN OUT  EFI_DEVICE_PATH_PROTOCOL **DevicePath
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>AtaPassThruGetDevice (
    IN  EFI_ATA_PASS_THRU_PROTOCOL *This,
    IN  EFI_DEVICE_PATH_PROTOCOL   *DevicePath,
    OUT UINT16                   *Port,
    OUT UINT16                   *PortMultiplierPort
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>AtaPassThruResetPort (
    IN EFI_ATA_PASS_THRU_PROTOCOL *This,
    IN UINT16                     Port
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>AtaPassThruResetDevice (
    IN EFI_ATA_PASS_THRU_PROTOCOL *This,
    IN UINT16                     Port,
    IN UINT16                     PortMultiplierPort
)
{
}

```

Example A-15—ATA Pass Thru Protocol implementation template

SimpleTextInput.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

#include "<<DriverName>>.h"

///
/// Simple Text Input Ex Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL g<<DriverName>>SimpleTextInputEx = {
    <<DriverName>>SimpleTextInputReset,
    <<DriverName>>SimpleTextInput.ReadKeyStrokeEx,
    NULL,
    <<DriverName>>SimpleTextInputSetState,
    <<DriverName>>SimpleTextInputRegisterKeyNotify,
    <<DriverName>>SimpleTextInputUnregisterKeyNotify
};

///
/// Simple Text Input Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_SIMPLE_TEXT_INPUT_PROTOCOL g<<DriverName>>SimpleTextInput = {
    (EFI_INPUT_RESET)<<DriverName>>SimpleTextInputReset,
    <<DriverName>>SimpleTextInput.ReadKeyStroke,
    NULL
};

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextInputReset (
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    IN BOOLEAN                               ExtendedVerification
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextInput.ReadKeyStrokeEx (
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    OUT EFI_KEY_DATA                      *KeyData
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextInputSetState (
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    IN EFI_KEY_TOGGLE_STATE            *KeyToggleState
)
{
}

EFI_STATUS

```

```

EFIAPI
<<DriverName>>SimpleTextInputRegisterKeyNotify (
    IN  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    IN  EFI_KEY_DATA                  *KeyData,
    IN  EFI_KEY_NOTIFY_FUNCTION      KeyNotificationFunction,
    OUT EFI_HANDLE                   *NotifyHandle
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextInputUnregisterKeyNotify (
    IN  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    IN  EFI_HANDLE                      NotificationHandle
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextInputReadKeyStroke (
    IN  EFI_SIMPLE_TEXT_INPUT_PROTOCOL   *This,
    OUT EFI_INPUT_KEY                  *Key
)
{
}

```

Example A-16—Simple Text Input Protocols implementation template

A.3.5

SimpleTextOutput.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

#include "<<DriverName>>.h"

///
/// Simple Text Output Protocol Mode instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_SIMPLE_TEXT_OUTPUT_MODE g<<DriverName>>SimpleTextOutputMode = {
    0,                                     // MaxMode
    0,                                     // Mode
    EFI_TEXT_ATTR (EFI_WHITE, EFI_BACKGROUND_BLACK), // Attribute
    0,                                     // CursorColumn
    0,                                     // CursorRow
    TRUE                                    // CursorVisible
};

///
/// Simple Text Output Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL g<<DriverName>>SimpleTextOutput = {
    <<DriverName>>SimpleTextOutputReset,
    <<DriverName>>SimpleTextOutputOutputString,
    <<DriverName>>SimpleTextOutputTestString,
};

```

```

<<DriverName>>SimpleTextOutputQueryMode,
<<DriverName>>SimpleTextOutputSetMode,
<<DriverName>>SimpleTextOutputSetAttribute,
<<DriverName>>SimpleTextOutputClearScreen,
<<DriverName>>SimpleTextOutputSetCursorPosition,
<<DriverName>>SimpleTextOutputEnableCursor,
&g<<DriverName>>SimpleTextOutputMode
};

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextOutputReset (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL      *This,
    IN BOOLEAN                               ExtendedVerification
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextOutputOutputString (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL      *This,
    IN CHAR16                                *String
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextOutputTestString (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL      *This,
    IN CHAR16                                *String
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextOutputQueryMode (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL      *This,
    IN UINTN                                 ModeNumber,
    OUT UINTN                                *Columns,
    OUT UINTN                                *Rows
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextOutputSetMode (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL      *This,
    IN UINTN                                 ModeNumber
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextOutputSetAttribute (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL      *This,
    IN UINTN                                 Attribute
)
{
}

EFI_STATUS
EFIAPI

```

```

<<DriverName>>SimpleTextOutputClearScreen (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL           *This
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextOutputSetCursorPosition (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL           *This,
    IN UINTN                                     Column,
    IN UINTN                                     Row
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleTextOutputEnableCursor (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL           *This,
    IN BOOLEAN                                    Visible
)
{
}

```

Example A-17—Simple Text Output Protocol implementation template

A.3.6

SerialIo.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

#include "<<DriverName>>.h"

///
/// Serial I/O Protocol Mode instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_SERIAL_IO_MODE g<<DriverName>>SerialIoMode = {
    0x00000000,          // ControlMask
    0,                   // Timeout
    0,                   // BaudRate
    0,                   // ReceiveF
    ifoDepth             // DataBits
    0,                   // Parity
    DefaultStopBits      // StopBits
};

///
/// Serial I/O Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_SERIAL_IO_PROTOCOL g<<DriverName>>SerialIo = {
    EFI_SERIAL_IO_PROTOCOL_REVISION,
    <<DriverName>>SerialIoReset,
    <<DriverName>>SerialIoSetAttributes,
    <<DriverName>>SerialIoSetControl,
};

```

```

<<DriverName>>SerialIoGetControl,
<<DriverName>>SerialIoWrite,
<<DriverName>>SerialIoRead,
&g<<DriverName>>SerialIoMode
};

EFI_STATUS
EFIAPI
<<DriverName>>SerialIoReset (
    IN EFI_SERIAL_IO_PROTOCOL *This
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SerialIoSetAttributes (
    IN EFI_SERIAL_IO_PROTOCOL      *This,
    IN UINT64                      BaudRate,
    IN UINT32                      ReceiveFifoDepth,
    IN UINT32                      Timeout,
    IN EFI_PARITY_TYPE             Parity,
    IN UINT8                       DataBits,
    IN EFI_STOP_BITS_TYPE          StopBits
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SerialIoSetControl (
    IN EFI_SERIAL_IO_PROTOCOL      *This,
    IN UINT32                      Control
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SerialIoGetControl (
    IN EFI_SERIAL_IO_PROTOCOL      *This,
    OUT UINT32                     *Control
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SerialIoWrite (
    IN EFI_SERIAL_IO_PROTOCOL      *This,
    IN OUT UINTN                   *BufferSize,
    IN VOID                        *Buffer
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SerialIoRead (
    IN EFI_SERIAL_IO_PROTOCOL      *This,
    IN OUT UINTN                   *BufferSize,
    OUT VOID                       *Buffer
)
{
}

```

Example A-18—Serial I/O Protocol implementation template

A.3.7

GraphicsOutput.c File

```
/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

#include "<<DriverName>>.h"

///
/// Graphics Output Protocol Mode structure
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE g<<DriverName>>GraphicsOutputMode = {
    0,           // MaxMode
    0,           // Mode
    NULL,        // Info
    0,           // SizeOfInfo
    0,           // FrameBufferBase
    0,           // FrameBufferSize
};

///
/// Graphics Output Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_GRAPHICS_OUTPUT_PROTOCOL g<<DriverName>>GraphicsOutput = {
    <<DriverName>>GraphicsOutputQueryMode,
    <<DriverName>>GraphicsOutputSetMode,
    <<DriverName>>GraphicsOutputBlt,
    &g<<DriverName>>GraphicsOutputMode
};

EFI_STATUS
EFIAPI
<<DriverName>>GraphicsOutputQueryMode (
    IN EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
    IN UINT32                      ModeNumber,
    OUT UINTN                       *SizeOfInfo,
    OUT EFI_GRAPHICS_OUTPUT_MODE_INFORMATION **Info
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>GraphicsOutputSetMode (
    IN EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
    IN UINT32                      ModeNumber
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>GraphicsOutputBlt (
    IN EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
    IN 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
)
{
}

```

Example A-19—Graphics Output Protocol implementation template

A.3.8

BlockIo.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

///
/// Block I/O Media structure
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_BLOCK_IO_MEDIA g<<DriverName>>BlockIoMedia = {
    0,           // MediaId
    FALSE,       // RemovableMedia
    FALSE,       // MediaPresent
    TRUE,        // LogicalPartition
    FALSE,       // ReadOnly
    FALSE,       // WriteCaching
    512,         // BlockSize
    0,           // IoAlign
    0,           // LastBlock
    0,           // LowestAlignedLba
    0,           // LogicalBlocksPerPhysicalBlock
    0            // OptimalTransferLengthGranularity
};

///
/// Block I/O Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_BLOCK_IO_PROTOCOL g<<DriverName>>BlockIo = {
    EFI_BLOCK_IO_PROTOCOL_REVISION3,          // Revision
    &g<<DriverName>>BlockIoMedia,           // Media
    (EFI_BLOCK_RESET)<<DriverName>>BlockIoReset, // Reset
    <<DriverName>>BlockIoReadBlocks,         // ReadBlocks
    <<DriverName>>BlockIoWriteBlocks,        // WriteBlocks
    <<DriverName>>BlockIoFlushBlocks        // FlushBlocks
};

///
/// Block I/O 2 Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_BLOCK_IO2_PROTOCOL g<<DriverName>>BlockIo2 = {
    &g<<DriverName>>BlockIoMedia,           // Media
    <<DriverName>>BlockIoReset,             // Reset
};

```

```

<<DriverName>>BlockIoReadBlocksEx,    // ReadBlocks
<<DriverName>>BlockIoWriteBlocksEx,   // WriteBlocks
<<DriverName>>BlockIoFlushBlocksEx   // FlushBlocks
};

///
/// Storage Security Command Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_STORAGE_SECURITY_COMMAND_PROTOCOL g<<DriverName>>StorageSecurityCommand = {
    <<DriverName>>StorageSecurityCommandReceiveData,
    <<DriverName>>StorageSecurityCommandSendData
};

EFI_STATUS
EFIAPI
<<DriverName>>BlockIoReadBlocks (
    IN EFI_BLOCK_IO_PROTOCOL           *This,
    IN UINT32                           MediaId,
    IN EFI_LBA                          Lba,
    IN UINTN                           BufferSize,
    OUT VOID                           *Buffer
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>BlockIoWriteBlocks (
    IN EFI_BLOCK_IO_PROTOCOL           *This,
    IN UINT32                           MediaId,
    IN EFI_LBA                          Lba,
    IN UINTN                           BufferSize,
    IN VOID                           *Buffer
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>BlockIoFlushBlocks (
    IN EFI_BLOCK_IO_PROTOCOL           *This
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>BlockIoReset (
    IN EFI_BLOCK_IO2_PROTOCOL          *This,
    IN BOOLEAN                         ExtendedVerification
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>BlockIoReadBlocksEx (
    IN     EFI_BLOCK_IO2_PROTOCOL *This,
    IN     UINT32                  MediaId,
    IN     EFI_LBA                 LBA,
    IN OUT EFI_BLOCK_IO2_TOKEN      *Token,
    IN     UINTN                  BufferSize,
    OUT    VOID                   *Buffer
)
{
}

```

```

EFI_STATUS
EFIAPI
<<DriverName>>BlockIoWriteBlocksEx (
    IN     EFI_BLOCK_IO2_PROTOCOL  *This,
    IN     UINT32                  MediaId,
    IN     EFI_LBA                 LBA,
    IN OUT EFI_BLOCK_IO2_TOKEN    *Token,
    IN     UINTN                  BufferSize,
    IN     VOID                   *Buffer
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>BlockIoFlushBlocksEx (
    IN     EFI_BLOCK_IO2_PROTOCOL  *This,
    IN OUT EFI_BLOCK_IO2_TOKEN    *Token
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>StorageSecurityCommandReceiveData (
    IN EFI_STORAGE_SECURITY_COMMAND_PROTOCOL  *This,
    IN UINT32                                     MediaId,
    IN UINT64                                     Timeout,
    IN UINT8                                      SecurityProtocolId,
    IN UINT16                                     SecurityProtocolSpecificData,
    IN UINTN                                      PayloadBufferSize,
    OUT VOID                                       *PayloadBuffer,
    OUT UINTN                                     *PayloadTransferSize
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>StorageSecurityCommandSendData (
    IN EFI_STORAGE_SECURITY_COMMAND_PROTOCOL  *This,
    IN UINT32                                     MediaId,
    IN UINT64                                     Timeout,
    IN UINT8                                      SecurityProtocolId,
    IN UINT16                                     SecurityProtocolSpecificData,
    IN UINTN                                      PayloadBufferSize,
    IN VOID                                       *PayloadBuffer
)
{
}

```

Example A-20—Block I/O, Block I/O 2, and Storage Security Protocols implementation template

A.3.9

NiiUndi.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

```

```

<<License>>

*/
#include "<<DriverName>>.h"

///
/// Network Interface Identifier Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL g<<DriverName>>Nii = {
    EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL_REVISION, // Revision
    0, // Id
    0, // ImageAddr
    0, // ImageSize
    { 'U', 'N', 'D', 'I' }, // StringId
    EfiNetworkInterfaceUndi, // Type
    PXE_ROMID_MAJORVER, // MajorVer
    PXE_ROMID_MINORVER, // MinorVer
    FALSE, // Ipv6Supported
    0 // IfNum
};

```

Example A-21—Network Interface Identifier Protocol implementation template

A.3.10

SimpleNetwork.c File

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

///
/// Simple Network Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_SIMPLE_NETWORK_PROTOCOL g<<DriverName>>SimpleNetwork = {
    EFI_SIMPLE_NETWORK_PROTOCOL_REVISION, // Revision
    <<DriverName>>SimpleNetworkStart, // Start
    <<DriverName>>SimpleNetworkStop, // Stop
    <<DriverName>>SimpleNetworkInitialize, // Initialize
    <<DriverName>>SimpleNetworkReset, // Reset
    <<DriverName>>SimpleNetworkShutdown, // Shutdown
    <<DriverName>>SimpleNetworkReceiveFilters, // ReceiveFilters
    <<DriverName>>SimpleNetworkStationAddress, // StationAddress
    <<DriverName>>SimpleNetworkStatistics, // Statistics
    <<DriverName>>SimpleNetworkMCastIpToMac, // MCastIpToMac
    <<DriverName>>SimpleNetworkNvData, // NvData
    <<DriverName>>SimpleNetworkGetStatus, // GetStatus
    <<DriverName>>SimpleNetworkTransmit, // Transmit
    <<DriverName>>SimpleNetworkReceive, // Receive
    NULL, // WaitForPacket
    NULL // Mode
};

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkStart (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This

```

```

        )
    }

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkStop (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkInitialize (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN UINTN                     ExtraRxBufferSize OPTIONAL,
    IN UINTN                     ExtraTxBufferSize OPTIONAL
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkReset (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN BOOLEAN                    ExtendedVerification
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkShutdown (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkReceiveFilters (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN UINT32                     Enable,
    IN UINT32                     Disable,
    IN BOOLEAN                    ResetMCastFilter,
    IN UINTN                      MCastFilterCnt OPTIONAL,
    IN EFI_MAC_ADDRESS            *MCastFilter OPTIONAL
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkStationAddress (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN BOOLEAN                    Reset,
    IN EFI_MAC_ADDRESS            *New OPTIONAL
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkStatistics (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN BOOLEAN                    Reset,
    IN OUT UINTN                 *StatisticsSize OPTIONAL,
    OUT EFI_NETWORK_STATISTICS   *StatisticsTable OPTIONAL
)
{
}

```

```

}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkMCastIpToMac (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN BOOLEAN                   IPv6,
    IN EFI_IP_ADDRESS             *IP,
    OUT EFI_MAC_ADDRESS           *MAC
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkNvData (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN BOOLEAN                   ReadWrite,
    IN UINTN                      Offset,
    IN UINTN                      BufferSize,
    IN OUT VOID                   *Buffer
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkGetStatus (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    OUT UINT32                   *InterruptStatus OPTIONAL,
    OUT VOID                      **TxBuf OPTIONAL
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkTransmit (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN UINTN                      HeaderSize,
    IN UINTN                      BufferSize,
    IN VOID                       *Buffer,
    IN EFI_MAC_ADDRESS             *SrcAddr OPTIONAL,
    IN EFI_MAC_ADDRESS             *DestAddr OPTIONAL,
    IN UINT16                     *Protocol OPTIONAL
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>SimpleNetworkReceive (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    OUT UINTN                     *HeaderSize OPTIONAL,
    IN OUT UINTN                  *BufferSize,
    OUT VOID                      *Buffer,
    OUT EFI_MAC_ADDRESS            *SrcAddr OPTIONAL,
    OUT EFI_MAC_ADDRESS            *DestAddr OPTIONAL,
    OUT UINT16                     *Protocol OPTIONAL
)
{
}

```

Example A-22—Simple Network Protocol implementation template

A.3.11

UserCredential.c File

```
/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

#include "<<DriverName>>.h"

///
/// User Credential Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_USER_CREDENTIAL2_PROTOCOL g<<DriverName>>UserCredential = {
    { 0x0, 0x0, 0x0, { 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 } }, // Identifier
    { 0x0, 0x0, 0x0, { 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 } }, // Type
    <<DriverName>>UserCredentialEnroll, // Enroll
    <<DriverName>>UserCredentialForm, // Form
    <<DriverName>>UserCredentialTile, // Tile
    <<DriverName>>UserCredentialTitle, // Title
    <<DriverName>>UserCredentialUser, // User
    <<DriverName>>UserCredentialSelect, // Select
    <<DriverName>>UserCredentialDeselect, // Deselect
    <<DriverName>>UserCredentialDefault, // Default
    <<DriverName>>UserCredentialGetInfo, // GetInfo
    <<DriverName>>UserCredentialGetNextInfo, // GetNextInfo
    0, // Capabilities
    <<DriverName>>UserCredentialDelete // Delete
};

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialEnroll (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    IN     EFI_USER_PROFILE_HANDLE       User
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialForm (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    OUT    EFI_HII_HANDLE                 *Hii,
    OUT    EFI_GUID                      *FormSetId,
    OUT    EFI_FORM_ID                   *FormId
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialTile (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    IN OUT   UINTN                        *Width,
    IN OUT   UINTN                        *Height,
    OUT     EFI_HII_HANDLE                 *Hii,
    OUT     EFI_IMAGE_ID                  *Image
)
{
}
```

```

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialTitle (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    OUT      EFI_HII_HANDLE                *Hii,
    OUT      EFI_STRING_ID                 *String
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialUser (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    IN       EFI_USER_PROFILE_HANDLE        User,
    OUT     EFI_USER_INFO_IDENTIFIER       *Identifier
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialSelect (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    OUT      EFI_CREDENTIAL_LOGON_FLAGS   *AutoLogon
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialDeselect (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialDefault (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL      *This,
    OUT     EFI_CREDENTIAL_LOGON_FLAGS          *AutoLogon
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialGetInfo (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    IN       EFI_USER_INFO_HANDLE           UserInfo,
    OUT     EFI_USER_INFO                  *Info,
    IN OUT    UINTN                      *InfoSize
)
{
}

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialGetNextInfo (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    IN OUT    EFI_USER_INFO_HANDLE         *UserInfo
)
{
}

```

```

EFI_STATUS
EFIAPI
<<DriverName>>UserCredentialDelete (
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    IN           EFI_USER_PROFILE_HANDLE     User
)
{
}

```

Example A-23—User Credential Protocol implementation template

A.3.12 *LoadFile.c File*

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

*/
#include "<<DriverName>>.h"

///
/// Load File Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_LOAD_FILE_PROTOCOL g<<DriverName>>LoadFile = {
    <<DriverName>>LoadFileLoadFile
};

EFI_STATUS
EFIAPI
<<DriverName>>LoadFileLoadFile (
    IN EFI_LOAD_FILE_PROTOCOL             *This,
    IN EFI_DEVICE_PATH_PROTOCOL          *FilePath,
    IN BOOLEAN                           BootPolicy,
    IN OUT UINTN                         *BufferSize,
    IN VOID                             *Buffer OPTIONAL
)
{
}

```

Example A-24—Load File Protocol implementation template

A.4 *Platform Specific UEFI Driver Templates*

This section contains templates for the implementation of protocols that are typically provided with the UEFI platform firmware.

A.4.1 *EdidOverride.c File*

```

/** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

```

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

///
/// EDID Override Protocol instance
///
GLOBAL_REMOVE_IF_UNREFERENCED
EFI_EDID_OVERRIDE_PROTOCOL g<<DriverName>>EdidOverride = {
    <<DriverName>>EdidOverrideGetEdid
};

EFI_STATUS
EFIAPI
<<DriverName>>EdidOverrideGetEdid (
    IN EFI_EDID_OVERRIDE_PROTOCOL
    IN EFI_HANDLE
    OUT UINT32
    IN OUT UINTN
    IN OUT UINT8
)
{
}
```

Example A-25—EDID Override Protocol implementation template

A.5

EDK II Package Extension Templates

This section contains templates for the extending the contents of an EDK II package. This is not a common operation, but some UEFI Driver implementations may choose to define new protocols, new GUIDs, or new library classes. This section also covers how to include protocols, GUIDs, and library classes in a UEFI Driver.

A.5.1

Protocol File Template

The .h files for protocols are placed in the include directories of EDK II packages. The typical path to a protocol .h file is

<<PackageName>>/Include/Protocol/<<ProtocolName>>.h. For example, all the protocols defined in the *UEFI Specification* can be found in the EDK II **MdePkg** in **/MdePkg/Include/Protocol**. When a new protocol is defined and added to an include directory of an EDK II package, the protocol must also be added to the **[Protocols]** section of a package's .dec file. The .dec file is where the C variable name for the protocol is declared and mapped to GUID value for the protocol. Defining a new protocol is not commonly required when implementing a new UEFI Driver. If a UEFI Driver implementation does require a new protocol definition, then the new protocol is usually added to the same EDK II package that contains the UEFI Driver implementation.

The example below shows a template for adding a new protocol to the **[Protocols]** section of an EDK II package .dec file. Example A-26 shows the template for the .h files for protocols placed in the include directory of an EDK II package.

```
[Protocols]
## Include/Protocol/<<ProtocolName>>.h
gEfi<<ProtocolName>>ProtocolGuid = <<GUID_STRUCT>>
```

Example A-26—Add protocol to an EDK II package

```
/** @file
<<BriefDescription>>
```

```

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/


#ifndef __<<PROTOCOL_NAME>>_H__
#define __<<PROTOCOL_NAME>>_H__


#define EFI_<<PROTOCOL_NAME>>_PROTOCOL_GUID \
<<GUID_STRUCT>>

/// Forward declaration
///
typedef struct _EFI_<<PROTOCOL_NAME>>_PROTOCOL EFI_<<PROTOCOL_NAME>>_PROTOCOL;

/// Function prototypes
///
typedef
EFI_STATUS
(EFIAPI *EFI_<<PROTOCOL_NAME>>_<<FUNCTION_NAME_1>>)(
    IN EFI_<<PROTOCOL_NAME>>_PROTOCOL *This
    //
    // Place additional function arguments here
    //
);

typedef
EFI_STATUS
(EFIAPI *EFI_<<PROTOCOL_NAME>>_<<FUNCTION_NAME_2>>)(
    IN EFI_<<PROTOCOL_NAME>>_PROTOCOL *This
    //
    // Place additional function arguments here
    //
);

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

/// Protocol structure
///
typedef struct _EFI_<<PROTOCOL_NAME>>_PROTOCOL {
    EFI_<<PROTOCOL_NAME>>_<<FUNCTION_NAME_1>>  <<FunctionName1>>;
    EFI_<<PROTOCOL_NAME>>_<<FUNCTION_NAME_2>>  <<FunctionName2>>;
    // . .
    EFI_<<PROTOCOL_NAME>>_<<FUNCTION_NAME_N>>  <<FunctionNameN>>;
    //
    // Place protocol data fields here
    //
} _EFI_<<PROTOCOL_NAME>>_PROTOCOL;

extern EFI_GUID gEfi<<ProtocolName>>ProtocolGuid;

#endif

```

Example A-27—Protocol include file template

A.5.2

GUID File Template

GUIDs and their associated data structures are declared just like protocols. The only difference is that GUIDs are typically placed in a different subdirectory of an EDK II package. The typical path to a protocol .h file is `<<PackageName>>/Include/Guid/<<GuidName>>.h`. For example, all the GUIDs defined in the *UEFI Specification* can be found in the EDK II `MdePkg` in `/MdePkg/Include/Guid`. When a new GUID is defined and added to an include directory of an EDK II package, the GUID must also be added to the `[Guids]` section of a package's .dec file. The .dec file is where the C variable name for the GUID is declared and mapped to GUID value for the protocol. Defining a new GUID is not commonly required when implementing a new UEFI Driver. If a UEFI Driver implementation does require a new GUID definition, then the new GUID is usually added to the same EDK II package that contains the UEFI Driver implementation.

The following example shows a template for adding a new GUID to the `[Guids]` section of an EDK II package .dec file. Following that, Example A-29 shows the template for the .h files for GUIDs placed in the include directory of an EDK II package.

```
[Guids]
## Include/Guid/<<GuidName>>.h
gEfi<<GuidName>>Guid = <<GUID_STRUCT>>
```

Example A-28—Add GUID to an EDK II package

```
** @file
<<BriefDescription>>

<<DetailedDescription>>

<<Copyright>>

<<License>>

**/

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


#define EFI_<<GUID_NAME>>_GUID \
    <<GUID_STRUCT>>

/// 
/// GUID specific defines
/// 

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

extern EFI_GUID gEfi<<GuidName>>Guid;

#endif
```

Example A-29—GUID include file template

A.5.3

Library Class File Template

Library Classes and their associated functions, defines, and data structures are declared very similar to protocols and GUIDs. The difference is that Library Classes are typically placed in a different subdirectory of an EDK II package. The typical path to a library .h file is <>PackageName>>/Include/Library/<>LibraryName>>.h. For example, all the libraries classes defined by the **MdePkg** can be found in the EDK II /MdePkg/Include/Library. When a new library class is defined and added to an include directory of an EDK II package, the library class must also be added to the [LibraryClasses] section of a package's .dec file. The .dec file is where the mapping between the name of the library class and the path to the include file for the library class is declared. Defining a new library class is not commonly required when implementing a new UEFI Driver. If a UEFI Driver implementation does require a new library class, then the new library class is usually added to the same EDK II package that contains the UEFI Driver implementation.

Example A-30 shows a template for adding a new library class to the [LibraryClasses] section of an EDK II package .dec file. Following that, Example A-31 shows the template for the .h files for a library class placed in the include directory of an EDK II package. All public functions provided by a library class must use the **EFI API** calling convention. The return type for a library class function is not required to be **EFI_STATUS**. **EFI_STATUS** is only shown in this template as an example.

```
[LibraryClasses]
## @libraryclass <>BriefDescription>>
##
<>LibraryClassName>> | Include/Library/<>LibraryClassName>>.h
```

Example A-30—Add Library Class to an EDK II package

```
/** @file
<>BriefDescription>>

<>DetailedDescription>>

<>Copyright>>

<>License>>

*/
#ifndef __<>LIBRARY_CLASS_NAME>>_H__
#define __<>LIBRARY_CLASS_NAME>>_H__

///
/// Library class public defines
///

///
/// Library class public structures/unions
///

///
/// Library class public functions
///
EFI_STATUS
EFIAPI
LibraryFunction1 (
    //
    // Additional function arguments here.
)
```

```

//  
);  
  
VOID  
EFIAPI  
LibraryFunction2 (  
//  
// Additional function arguments here.  
//  
);  
  
UINT8  
EFIAPI  
LibraryFunction3 (  
//  
// Additional function arguments here.  
//  
);  
  
#endif

```

Example A-31—Library Class include file template

A.5.4

Including Protocols, GUIDs, and Library Classes

A UEFI Driver that produces or consumes a protocol or GUID must include the protocol or GUID definitions using `#include` statements and also declare the usage of those Protocols or GUIDs in the `[Protocols]` and `[Guids]` sections of the INF file for the UEFI Driver. A UEFI Driver that uses defines, structures, unions, or functions from a library class must include the those definitions using a `#include` statement.

The `#include` statements use paths in EDK II packages to the .h files for the required protocols or GUIDs or library classes. The include paths that an EDK II package supports are declared in the `[Includes]` section of an EDK II package. An EDK II package typically uses an include directory called `Include` and use subdirectories called `Protocol` and `Guid` and `Library` for .h files. The example below shows some example `#include` statements for a UEFI Driver that uses protocols and GUIDs and library classes from the `MdePkg`. The `MdePkg` contains all the protocols and GUIDs defined in the *UEFI Specification* along with a number of library classes that are very useful to UEFI Drivers. The include file `Uefi.h` pulls in all the standard definitions from the *UEFI Specification*. This file must be included before any other include files. This next three include statements pull in the Block I/O Protocol, the Driver Binding Protocol, and the Component Name 2 Protocol. The next 2 include statements pull in the definitions for the UEFI Global Variable GUID and the GUID associated with the SMBIOS table that may be registered in the UEFI System Table. The last set of include statements pull in the definitions from a number of library classes that are commonly used by UEFI Drivers.

```

#include <Uefi.h>  
  
#include <Protocol/DriverBinding.h>  
#include <Protocol/ComponentName2.h>  
#include <Protocol/BlockIo.h>  
  
#include <Guid/GlobalVariable.h>  
#include <Guid/Smbios.h>

```

```
#include <Library/UefiBootServicesTableLib.h>
#include <Library/MemoryAllocationLib.h>
#include <Library/BaseMemoryLib.h>
#include <Library/BaseLib.h>
#include <Library/UefiLib.h>
#include <Library/DebugLib.h>
```

Example A-32—Protocol, GUID, and Library Class include statements

Example A-33, below, shows a portion of an INF file for the same UEFI Driver that requires the protocols and GUIDs included in Example A-32 above. A UEFI Driver must declare the Protocols, GUIDs, and Library Classes the UEFI Driver uses in the INF file. The comment blocks associated with each Protocol and GUID are optional, but they describe how each Protocol and GUID is used by the UEFI Driver. This specific example shows that this UEFI Driver produces the Driver Binding Protocol and the Component Name Protocol, and it consumes the Block I/O Protocol in the `start()` function of the Driver Binding Protocol. It also shows that UEFI Global Variable GUID is used to access the variable called `ConIn` and that the SMBIOS Table GUID is used to lookup the SMBIOS Table in the UEFI System Table.

```
[LibraryClasses]
UefiBootServicesTableLib
MemoryAllocationLib
BaseMemoryLib
BaseLib
UefiLib
DebugLib

[Protocols]
gEfiDriverBindingProtocolGuid      ## PRODUCES
gEfiComponentName2ProtocolGuid    ## PRODUCES
gEfiBlockIoProtocolGuid          ## TO_START

[Guids]
gEfiGlobalVariableGuid  ## CONSUMES ## Variable:L"ConIn"
gEfiSmbiosTableGuid       ## CONSUMES
```

Example A-33—Protocol and GUID INF statements

Appendix B ***EDK II Sample Drivers***

This appendix lists sample UEFI Drivers in the EDK II open source project along with their UEFI Driver related properties. This is not an exhaustive list of UEFI Drivers available from the EDK II open source project. Instead, a set of UEFI Drivers that provide examples of each major UEFI Driver feature this guide describes are listed. This appendix may be used to review UEFI Driver implementations that implement a specific UEFI Driver feature. Or it may be used to find an example UEFI Driver with a feature set that closely matches the features required for a new UEFI Driver so an existing driver can be cloned as a starting point.

Table 47—UEFI Driver Properties

Field	Field value	Description
DB		Number of Driver Binding Protocols installed in the driver entry point.
CFG	1	Driver Configuration Protocol is installed in the driver entry point.
	2	Driver Configuration 2 Protocol is installed in the driver entry point.
	*	Both Driver Configuration and Driver Configuration 2 are installed.
	H	HII packages are installed in the driver entry point for configuration.
DIAG	1	Driver Diagnostics Protocol is installed in the driver entry point.
	2	Driver Diagnostics 2 Protocol is installed in the driver entry point.
	*	Both Driver Diagnostics and Driver Diagnostics 2 are installed.
CN	1	Component Name Protocol is installed in the driver entry point.
	2	Component Name 2 Protocol is installed in the driver entry point.
	*	Both Component Name and Component Name 2 are installed.
Class	B	Bus driver.
	D	Device driver.
	H	Hybrid driver.
	R	Root bridge driver.
	S	Service driver.
Child	I	Initializing driver.
	All	All child handles in first call to Start().
Child	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

Field	Field value	Description
Type	B	UEFI Boot Services Driver.
	R	UEFI Runtime Driver.
UL	Y	Driver is unloadable.
HP	Y	Driver supports a Hot Plug device or bus

Table 48—Sample UEFI Driver Properties

Driver	D B	C F G	D I A G	C N	C L A S S	C H I L D	P A R E N T	T Y P E	U L	H P
IntelFrameworkModulePkg/Bus/Isa/ IsaBusDxe	1	-	-	*	B	All	1	B	-	-
IntelFrameworkModulePkg/Bus/Isa/ IsaFloppyDxe	1	-	-	*	D	-	1	B	-	-
IntelFrameworkModulePkg/Bus/Isa/ IsaSerialDxe	1	-	-	*	B	1	1	B	-	-
IntelFrameworkModulePkg/Bus/Isa/ Ps2KeyboardDxe	1	-	-	*	D	-	1	B	-	-
IntelFrameworkModulePkg/Bus/Isa/ Ps2MouseDxe	1	-	-	*	D	-	1	B	-	-
MdeModulePkg/Bus/Pci/ PciBusDxe	1	-	-	*	B	1/ All	1	B	-	Y
DuetPkg/ PciBusNoEnumerationDxe	1	-	-	*	B	1/ All	1	B	-	-
MdeModulePkg/Bus/Pci/ UhciDxe	1	-	-	*	D	-	1	B	-	-
MdeModulePkg/Bus/Pci/ EhciDxe	1	-	-	*	D	-	1	B	-	-
MdeModulePkg/Bus/Pci/ XhciDxe	1	-	-	*	D	-	1	B	-	-
OptionRomPkg/ UndiRuntimeDxe	1	-	-	-	B	1	1	R	-	-
OptionRomPkg/ CirrusLogic5430Dxe	1	-	-	*	D	-	1	B	-	-
MdeModulePkg/Bus/Scsi/ ScsiBusDxe	1	-	-	*	B	1/ All	1	B	-	-
MdeModulePkg/Bus/Scsi/ ScsiDiskDxe	1	-	-	*	D	-	1	B	-	-
MdeModulePkg/Bus/Ata/ AtaAtapiPassThruDxe	1	-	-	*	D	-	1	B	-	-
MdeModulePkg/Bus/Ata/ AtaBusDxe	1	-	-	*	B	1/ All	1	B	-	-
MdeModulePkg/Bus/Usb/ UsbBusDxe	1	-	-	*	B	0	1	B	-	Y
MdeModulePkg/Bus/Usb/ UsbKbDxe	1	-	-	*	D	-	1	B	-	-

Driver	D B	C F G	D I A G	C N	C L A S S	C H I L D	P A R E N T	T Y P E	U L	H P
MdeModulePkg/Bus/Usb/ UsbMassStorageDxe	1	-	-	*	D	-	1	B	-	-
MdeModulePkg/Bus/Usb/ UsbMouseDxe	1	-	-	*	D	-	1	B	-	-
IntelFrameworkModulePkg/Bus/Pci/ IdeBusDxe	1	1	*	*	H	1/ All	1	B	-	-
FatPkg/ EnhancedFatDxe	1	-	-	*	D	-	1	B	Y	-
MdeModulePkg/Universal/Console/ ConPlatformDxe	2	-	-	*	D	-	1	B	-	-
MdeModulePkg/Universal/Console/ ConSplitterDxe	4	-	-	*	B	All	> 1	B	-	-
MdeModulePkg/Universal/Console/ TerminalDxe	1	-	-	*	H	1	1	B	-	-
MdeModulePkg/Universal/ EbcDxe	-	-	-	-	S	-	-	B	-	-
PcAtChipsetPkg/ PciHostBridgeDxe	-	-	-	-	R	-	-	B	-	-
MdeModulePkg/Universal/Network/ Ip4ConfigDxe	1	H	-	*	H	All	1	B	Y	-
MdeModulePkg/Universal/ HiiResourcesSampleDxe	-	H	-	-	S	-	-	B	Y	-

Appendix C

Glossary

The following table defines terms used in this document. See the glossary in the *UEFI Specification* for definitions of additional terms.

Table 49—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 directly loads 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.

Term	Definition
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.
IA32	32-bit Intel architecture.
IBV	Independent BIOS vendor.

Term	Definition
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 IA32, 8 bytes on X64 and IPF)
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.

Term	Definition
PNPID	Plug and Play ID.
POST	Power On Self Test.
PPP	Point-to-Point Protocol.
PUN	Physical Unit Number.
PEI	Pre-boot 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.