
THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES
WINDOWS 7
DEVICE DRIVER
RONALD D. REEVES

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Reeves, Ron.
Windows 7 device driver / Ronald D. Reeves.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-67021-2 (pbk. : alk. paper)
ISBN-10: 0-321-67021-3 (pbk. : alk. paper)
1. Microsoft Windows device drivers (Computer programs)
I. Title.
QA76.76.D49R44 2011
005.7'1—dc22
2010039109

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-67021-2
ISBN-10: 0-321-67021-3

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, November 2010

CONTENTS

Preface xv
About the Author xix

Introduction **1**

PART I DEVICE DRIVER ARCHITECTURE OVERVIEW **5**

Chapter 1 Objects **7**

1.1 Nature of an Object 7
1.2 What Is a Software Object? 8
1.3 Gaining an Understanding 10
1.4 Software Components 11

Chapter 2 Windows Driver Foundation (WDF) Architecture **13**

2.1 WDF Component Functions 13
2.2 Design Goals for WDF 14
2.3 Device and Driver Support in WDF 15
2.4 WDF Driver Model 16
2.5 WDF Object Model 17
   2.5.1 Kernel Mode Objects 19
   2.5.2 User Mode Objects 19
2.6 Plug and Play and Power Management Support 20
   2.6.1 Plug and Play/Power Management State Machine 21
2.7 Integrated I/O Queuing and Cancellation 22
   2.7.1 Concurrency 22
   2.7.2 I/O Model 23
   2.7.3 I/O Request Flow 24
   2.7.4 Device I/O Requests 25
   2.7.5 Plug and Play and Power Management Requests 26

2.8 WMI Requests (Kernel Mode Drivers Only) 27
2.9 Driver Frameworks 28
2.9.1 Kernel Mode Framework 29
2.9.2 User Mode Framework 31
2.10 Windows Kernel 32
2.10.1 Reflector 32
2.10.2 Driver Host Process 32
2.10.3 Driver Manager 33
2.11 Tools for Development and Testing 33
2.11.1 PREfast for Drivers 34
2.11.2 Static Driver Verification (SDV) 35
2.11.3 Frameworks Verifier 36
2.11.4 Trace Logging 36
2.11.5 Debugger Extensions 37
2.11.6 Serviceability and Versioning 37

PART II USER MODE DRIVERS 39

Chapter 3 Windows 7 User Mode Drivers Overview and Operation 41

3.1 Devices Supported in User Mode 42
3.2 UMDF Model Overview 43
3.2.1 UMDF Object Model 45
3.2.2 UMDF Objects 45
3.3 Driver Callback Interfaces 47
3.4 UMDF Driver Features 49
3.4.1 Impersonation 50
3.4.2 Device Property Store 50
3.5 I/O Request Flow 51
3.5.1 I/O Request Dispatching 53
3.5.2 Create, Cleanup, and Close Requests 53
3.5.3 Create, Read, Write, and Device I/O Control Requests . . 56
3.6 I/O Queues 56
3.6.1 Dispatch Type 58
3.6.2 Queues and Power Management 59
3.7 I/O Request Objects 60
3.7.1 Retrieving Buffers from I/O Requests 61
3.7.2 Sending I/O Requests to an I/O Target 61
3.7.3 Creating Buffers for I/O Requests 63

3.7.4 Canceled and Suspended Requests 64
3.7.5 Completing I/O Requests 66
3.7.6 Adaptive Time-Outs 66
3.8 Self-Managed I/O 67
3.9 Synchronization Issues 68
3.10 Locks 70
3.11 Plug and Play and Power Management Notification 70
3.12 Device Enumeration and Startup 71
3.13 Device Power-Down and Removal 72
3.13.1 Surprise-Removal Sequence 74
3.14 Build, Test, and Debug 75
3.14.1 Installation and Configuration 76
3.14.2 Versioning and Updates 77

Chapter 4 Programming Drivers for the User Mode Driver Framework 79

4.1 Windows I/O Overview 79
4.2 Brief COM Information 81
4.3 UMDF Architecture 82
4.4 Required Driver Functionality 84
4.5 UMDF Sample Drivers 87
4.5.1 Minimal UMDF Driver: The Skeleton Driver 88
4.5.2 Skeleton Driver Classes, Objects, and Interfaces 89
4.6 Driver Dynamic-Link Library and Exports 91
4.6.1 Driver Entry Point: DllMain 91
4.6.2 Get Class Object: DllGetClassObject 93
4.7 Functions for COM Support 95
4.7.1 IUnknown Methods 95
4.7.2 IClassFactory Interface 96
4.7.3 Driver Callback Object 96
4.7.4 Device Callback Object 100
4.8 Using the Skeleton Driver as a Basis for Development 106
4.8.1 Customize the Exports File 107
4.8.2 Customize the Sources File 107
4.8.3 Customize the INX File 108
4.8.4 Customize the Comsup.cpp File 108
4.8.5 Add Device-Specific Code to Driver.cpp 109
4.8.6 Add Device-Specific Code to Device.cpp 109

Chapter 5 Using COM to Develop UMDF Drivers 111

5.1 Getting Started 111
5.1.1 COM Fundamentals 112
5.1.2 HRESULT 114
5.2 Using UMDF COM Objects 116
5.2.1 Obtaining an Interface on a UMDF Object 117
5.2.2 Reference Counting 119
5.3 Basic Infrastructure Implementation 120
5.3.1 DllMain 121
5.3.2 DllGetClassObject 121
5.3.3 Driver Object's Class Factory 122
5.3.4 Implementing a UMDF Callback Object 122
5.3.5 Implementing QueryInterface 125

PART III KERNEL MODE DRIVERS 127

Chapter 6 Windows 7 Kernel Mode Drivers Overview and Operations 129

6.1 KMDF Supported Devices 129
6.2 KMDF Components 131
6.3 KMDF Driver Structure 132
6.4 Comparing KMDF and WDM Drivers .132
6.5 Device Objects and Driver Roles 135
6.5.1 Filter Drivers and Filter Device Objects 136
6.5.2 Function Drivers and Functional Device Objects 136
6.5.3 Bus Drivers and Physical Device Objects 137
6.5.4 Legacy Device Drivers and Control Device Objects 138
6.6 KMDF Object Model 139
6.6.1 Methods, Properties, and Events 139
6.6.2 Object Hierarchy 141
6.6.3 Object Attributes 144
6.6.4 Object Context 145
6.6.5 Object Creation and Deletion 146
6.7 KMDF I/O Model 147
6.7.1 I/O Request Handler 149
6.7.2 I/O Queues 152
6.7.3 I/O Request Objects 154
6.7.4 Retrieving Buffers from I/O Requests 155

6.7.5 I/O Targets 156
6.7.6 Creating Buffers for I/O Requests 157
6.7.7 Canceled and Suspended Requests 158
6.7.8 Completing I/O Requests 160
6.7.9 Self-Managed I/O 161
6.7.10 Accessing IRPs and WDM Structures 161

Chapter 7 Plug and Play and Power Management 163

7.1 Plug and Play and Power Management Overview 163
7.2 Device Enumeration and Startup 164
7.2.1 Startup Sequence for a Function or Filter Device Object . . .165
7.2.2 Startup Sequence for a Physical Device Object 166
7.2.3 Device Power-Down and Removal 167
7.3 WMI Request Handler 172
7.4 Synchronization Issues 173
7.4.1 Synchronization Scope 175
7.4.2 Execution Level 177
7.4.3 Locks 178
7.4.4 Interaction of Synchronization Mechanisms 179
7.5 Security 180
7.5.1 Safe Defaults 180
7.5.2 Parameter Validation 180
7.5.3 Counted UNICODE Strings 181
7.5.4 Safe Device Naming Techniques 181

Chapter 8 Kernel Mode Installation and Build 183

8.1 WDK Build Tools 183
8.2 Build Environment 185
8.3 Building a Project 186
8.4 Building Featured Toaster 187
8.4.1 Makefile and Makefile.inc 187
8.4.2 The Sources File 188
8.4.3 The Build 190
8.5 Installing a KMDF Driver 190
8.5.1 The WDF Co-Installer 191
8.5.2 The INF 191
8.5.3 INFs for KMDF Drivers 192
8.5.4 wdffeatured.inf 192

8.6 Catalog Files and Digital Signature 193
8.7 Installing Featured Toaster 194
8.8 Testing a KMDF Driver 196
8.8.1 PREfast 196
8.8.2 Static Driver Verifier 197
8.8.3 KMDF Log 198
8.8.4 KMDF Verifier 198
8.8.5 Debugging a KMDF Driver 198
8.8.6 Kernel Debugging 200
8.8.7 KMDF Driver Features 201
8.9 Debugging Macros and Routines 203
8.10 WDF Debugger Extension Commands 204
8.11 Using WPP Tracing with a KMDF Driver 205
8.12 Using WinDbg with Featured Toaster 205
8.13 Versioning and Dynamic Binding 208

Chapter 9 Programming Drivers for the Kernel Mode Driver Framework 211

9.1 Differences Between KMDF and WDM Samples 216
9.2 Macros Used in KMDF Samples 218
9.3 KMDF Driver Structure and Concepts 219
9.3.1 Object Creation 220
9.3.2 Object Context Area 221
9.3.3 I/O Queues 222
9.3.4 I/O Requests 224
9.4 A Minimal KMDF Driver: The Simple Toaster 224
9.4.1 Creating a WDF Driver Object: DriverEntry 225
9.4.2 Creating the Device Object, Device Interface, and I/O Queue: EvtDriverDeviceAdd 227
9.4.3 Device Object and Device Context Area 229
9.4.4 Device Interface 231
9.4.5 Default I/O Queue 232
9.4.6 Handling I/O Request: EvtIoRead, EvtIoWrite, EvtIoDevice Control 233
9.5 Sample Software-Only Driver 235
9.5.1 File Create and Close Requests 235
9.5.2 Additional Device Object Attributes 237
9.5.3 Setting Additional Device Object Attributes 240

Chapter 10 Programming Plug and Play and Power Management 243

10.1 Registering Callbacks 243
10.1.1 Sample Code to Register Plug and Play and Power Callbacks 245
10.2 Managing Power Policy 248
10.2.1 Code to Set Power Policy 249
10.3 Callbacks for Power-Up and Power-Down 250
10.4 Callback for Wake Signal Support 251

Chapter 11 Programming WMI Support 253

11.1 WMI Architecture 253
11.2 Registering as a WMI Data Provider 254
11.3 Handling WMI Requests 255
11.4 WMI Requirements for WDM Drivers 256
11.5 WMI Class Names and Base Classes 257
11.6 Firing WMI Events 260
11.7 Troubleshooting Specific WMI Problems 265
11.7.1 Driver's WMI Classes Do Not Appear in the \root\wmi NameSpace 265
11.7.2 Driver's WMI Properties or Methods Cannot Be Accessed 266
11.7.3 Driver's WMI Events Are Not Being Received 267
11.7.4 Changes in Security Settings for WMI Requests Do Not Take Effect 267
11.8 Techniques for Testing WMI Driver Support 268
11.8.1 WMI IRPs and the System Event Log 269
11.8.2 WMI WDM Provider Log 269
11.9 WMI Event Tracing 269

Chapter 12 Programming KMDF Hardware Driver 273

12.1 Support Device Interrupts 274
12.1.1 Creating an Interrupt Object 274
12.1.2 Code to Create an Interrupt Object 275
12.1.3 Enabling and Disabling Interrupts 276
12.1.4 Code to Enable Interrupts 276
12.1.5 Code to Disable Interrupts 277

12.1.6 Post-Interrupt Enable and Pre-Interrupt Disable Processing . . . 277
12.2 Handling Interrupts . . . 278
12.2.1 Code for EvtInterruptIsr Callback . . . 279
12.2.2 Deferred Processing for Interrupts . . . 281
12.3 Mapping Resources . . . 283
12.3.1 Code to Map Resources . . . 284
12.3.2 Code to Unmap Resources . . . 288

Chapter 13 Programming Multiple I/O Queues and Programming I/O . . . 291

13.1 Introduction to Programming I/O Queues . . . 291
13.2 Creating and Configuring the Queues . . . 293
13.2.1 Code to Create Queues for Write Requests . . . 294
13.2.2 Code to Create Queues for Read Requests . . . 296
13.2.3 Code to Create Queues for Device I/O Control Requests . . . 297
13.3 Handling Requests from a Parallel Queue . . . 298
13.3.1 Code to Handle I/O Requests . . . 299
13.3.2 Performing Buffered I/O . . . 301
13.4 Forwarding Requests to a Queue . . . 302
13.5 Retrieving Requests from a Manual Queue . . . 303
13.5.1 Code to Find a Request . . . 304
13.6 Reading and Writing the Registry . . . 308
13.6.1 Code to Read and Write the Registry . . . 309
13.7 Watchdog Timer: Self-Managed I/O . . . 312
13.7.1 Self-Managed I/O Device Startup and Restart . . . 313
13.7.2 Self-Managed I/O During Device Power-Down and Removal . . . 314
13.7.3 Implementing a Watchdog Timer . . . 315

Appendix Driver Information Web Sites . . . 323

Bibliography . . . 331

Index . . . 333

PREFACE

This book provides the technical guidance and understanding needed to write device drivers for the new Windows 7 Operating System. It takes this very complex programming development, and shows how the Windows Driver Framework has greatly simplified this undertaking. It explains the hardware and software architecture you must understand as a driver developer. However, it focuses this around the actual development steps one must take to develop one or the other of the two types of drivers. Thus, this book's approach is a very pragmatic one in that it explains the various software APIs and computer and device hardware based upon our actual device handler development.

There has been great progress in the art of creating and debugging device drivers. There is now a great deal of object-oriented design techniques associated with the driver frameworks that are available to the device driver developer. Much of the previous grunt work, thank goodness, is now being handled by the latest device development framework Windows Driver Foundation (WDF). We will be covering both the user mode and kernel mode of device driver development. WDF has excellent submodels contained within it, called the User Mode Driver Framework and the Kernel Mode Driver Framework.

It is really great to see a Windows Driver Framework involved in the creation of Windows Device Drivers. I started working with Windows in 1990 and we primarily used the Win32 System APIs to communicate and control the Windows Operating System for our applications. We used the Device Driver Kit (DDK) to create the Windows drivers. Because I had my own company to create application software, I obviously was very concerned about the time it took to develop application software, and the robustness of the application. There were more than 2,000 Win32 APIs to be used for this task.

Then in about 1992, Microsoft came out with the Microsoft Framework Classes (MFC). In these 600+ classes, most of the Win32 APIs were encapsulated. Of course, prior to this, around 1988, the C++ compiler came out, and Object Oriented Programming started to come into its own. By using the MFC Framework, we could produce more application software faster and with better quality. My return on investment (ROI) went up, and I made more money. This sure made a believer of me in the use of frameworks. I used MFC until the .NET Framework came out, and for the last nine years I have been using this great collection of classes. All along, Microsoft was working to bring this same kind of software development improvements to developing device drivers. We came from the DDK, to the Windows Driver Model, to the Windows Driver Foundation Framework.

Therefore, this book shows how to create Windows 7 Device Drivers using the Windows Driver Foundation Framework. This should give us driver developers a little more sanity when meeting our deadlines.

The book is broken into three major parts as follows:

- **Part I, "Device Driver Architecture Overview"**—This part lays out the architecture involved in both software and hardware for device handler development. It also covers the driver development environment needed for driver development, for both types of drivers that are normally developed—that is, User Mode and Drivers. This section also covers the two Windows driver frameworks that are most commonly used for driver device development today, which are part of the Windows Driver Framework (WDF). These two Windows Driver Frameworks are the User Mode Driver Framework (UMDF) and the Kernel Mode Driver Framework (KMDF).

- **Part II, "User Mode Drivers"**—This part outlines the approach, design, development, and debug of User Mode Drivers. This part takes the driver programmer from start to finish in developing User Mode Drivers. We primarily use the User Mode Driver Framework for all of this work. The code is done in C++ because it is the best way to develop these types of drivers. Discussions are based on a USB User Mode Driver that we will develop using the UMDF. We will use a USB hardware learning kit from Open Systems Resources, Inc. (OSR). This provides a hardware simulation to test our User Mode Drivers. This part is primarily stand-alone and could be read and used without reading any other parts of the book. However, you will probably want to read Part I to get a feel for what we are using.

- **Part III, "Kernel Mode Drivers"**—This part outlines the approach, design, development, and debug of Kernel Mode Drivers. The intent again is to take the driver programmer from start to finish in developing Kernel Mode Drivers. For this section, we primarily use the Kernel Mode Driver Framework for all of this work. The code is done in C because this is the best way to develop these types of drivers. Discussions are based on a Kernel Mode Driver that we develop using the KMDF. We use a Peripheral Component Interconnect (PCI) hardware learning kit from OSR. This provides a hardware simulation to test our Kernel Mode Drivers. The section is also primarily stand-alone and could be read and used without reading any other parts of the book. Again, you will probably want to read Part I to get a feel for what we are using.

ACKNOWLEDGMENTS

I am most grateful to my editor Bernard Goodwin at Pearson Education for giving me the opportunity to write this book. His support during the preparation was great. I would also like to thank his assistant Michelle Housley for her timely fashion in getting me reference books and material. Also, I would like to thank John Herrin, Video Project Manager at Pearson Education, for support and help in creating the book video. Thanks to Michael Thurston, my development editor, for making the book sound very polished.

This page intentionally left blank

INTRODUCTION

Device drivers are where the rubber meets the road, and are very specialized pieces of software that allow your application programs to communicate to the outside world. Any communications your Windows 7 makes to the outside world requires a Device Driver. These devices include such things as mouse, display, keyboard, CD-ROMS, data acquisition, data network communication, and printers. However, Microsoft has written and supplied a great many drivers with the Windows 7 Operating System. These drivers support most of what we call the standard devices, and we will not be covering them in this book.

This book is about how we create device drivers for the nonstandard devices—devices that are not typically found on standard PCs. Quite often, the market is too small for Microsoft to create a standard device driver for these types of devices—such things as data acquisition boards, laboratory equipment, special test equipment, and communications boards.

This discussion will highlight the significant features of interest to the device driver developers. Figure I.1 shows a general block diagram of Windows 7. We develop more detailed block diagrams in the discussions in various parts of the book.

In Figure I.1 the user applications don't call the Windows 7 Operating System Services directly. They go thru the Win32 subsystem dynamic-linked libraries (DLL). The User Mode Device Drivers, discussed later, go through this same communication channel.

The various Windows 7 services that run independently are handled by the Service Processes. They are typically started by the service control manager.

The various Windows 7 System Support Processes are not considered Windows 7 services. They are therefore not started by the service control manager.

The Windows 7 I/O Manager actually consists of several executive subsystems that manage hardware devices, priority interfaces for both the system and the applications. We cover this in detail in Parts II and III of this book.

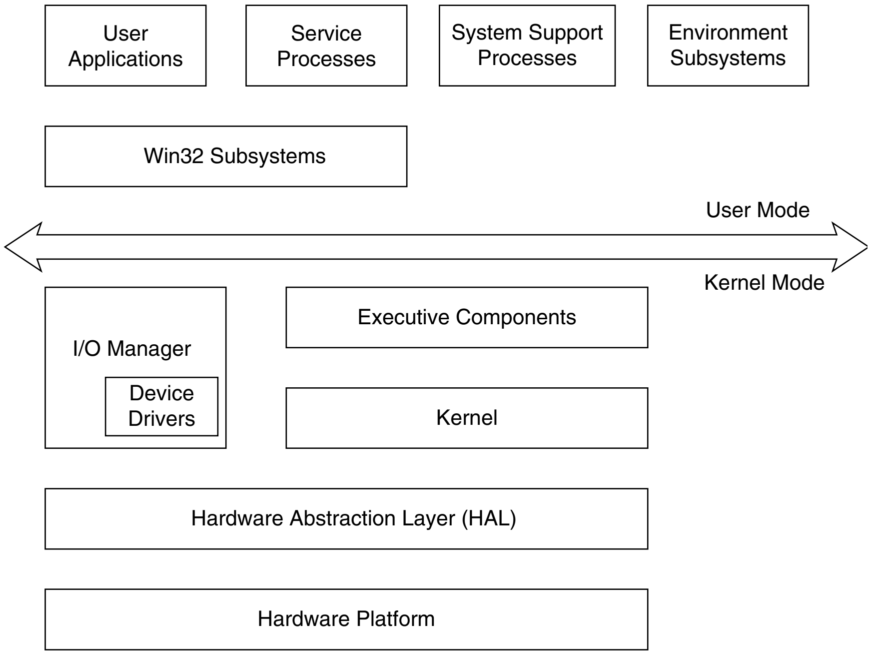


Figure I.1 System Overview Windows 7

The Device Driver block shown in the I/O Manager block is primarily what this book is all about—that is, designing, developing, and testing Windows 7 Device Drivers. The drivers of course translate user I/O function calls into hardware device I/O requests.

The Hardware Abstraction Layer (HAL) is a layer of code that isolates platform-specific hardware differences from the Windows 7 Operating System. This allows the Windows 7 Operating System to run on different hardware motherboards. When device driver code is ported to a new platform, in general, only a recompile is necessary. The device driver code relies on code (macros) within HAL to reference hardware buses and registers. HAL usage in general is implemented such that inline performance is achieved.

The Windows 7 performance goals often impact device driver writers. When system threads and users request service from a device, it's very important that the driver code not block execution. In this case, where the driver request cannot be handled immediately, the request must be queued for subsequent handling. As we will show in later discussions, the I/O Manager routines available allow us to do this.

Windows 7 gives us a rich architecture for applications to utilize. However, this richness has a price that device driver authors often have to pay. Microsoft, realizing this early on some 14 years ago, started developing the driver development models and framework to aid the device driver author. The earliest model, the Windows Driver Model (WDM) had a steep learning curve, but was a good step forward. Microsoft has subsequently developed the Windows Driver Foundation (WDF) that makes developing robust Windows 7 drivers easier to implement and learn. This book is about developing Windows 7 Device Driver using WDF.

This page intentionally left blank

This page intentionally left blank

CHAPTER 1

OBJECTS

Before we go into the discussion on drivers, we need to first briefly review objects, which are mentioned extensively throughout the book.

1.1 Nature of an Object

One of the fundamental ideas in software component engineering is the use of objects. But just what is an object? There doesn't seem to be a universally accepted idea as to what an object is. The view that the computer scientist Grady Booch (1991) takes is that an object is defined primarily by three characteristics: its state, its behavior, and its identity. The fundamental unit of analysis, in most cognitive theories, is the information-processing component. A component is an elementary information process that operates on the internal representation of objects or symbols (Newell & Simon 1972; Sternberg 1977). If we look at the way these components work, they may translate a sensory input into a conceptual representation, transform one conceptual representation into another, or translate a conceptual representation into a motor output.

The Object Oriented Programming (OOP) techniques for software have been around now for approximately a quarter of a century. But the phenomenon is not new. Ancient philosophers, such as Plato and Aristotle, as well as modern philosophers like Immanuel Kant have been involved in explaining the meaning of existence in general and determining the essential characteristics of concepts and objects (Rand 1990). Very recently Minsky developed a theory of objects, whose behavior closely resembles processes that take place in the human mind (Minsky 1986). Novak and Gowin (Novak and Gowin 1984) showed how objects play an important role in education and cognitive science. Their approach is one in which concepts are discovered by finding patterns in objects designated by some name. But wait, we were talking about objects and now we are talking about concepts. That is because concepts reflect the way we divide the world into classes, and much of what we learn, communicate, and reason about involves relations among these classes. Concepts are mental representations of classes, and their salient function is to promote cognitive economy. A class then can be seen as a template for generating objects with similar structure and behavior.

The Object Management Group (OMG) defines a class as follows:

> A class is an implementation that can be instantiated to create multiple objects with the same behavior. An object is an instance of a class.

From the software point of view, by partitioning the software into classes, we decrease the amount of information we must perceive, learn, remember, communicate, and reason about.

1.2 What Is a Software Object?

What is a software object? In 1976, Niklaus Wirth published his book *Algorithms + Data Structures = Programs.* The relationship of these two aspects heightens our awareness of the major parts of a program. In 1986, J. Craig Cleaveland published his book *Data Types*. In 1979 Bjarne Stroustrup had started the work on C with classes. By 1985, the C++ Programming Language had evolved and in 1990 the book *The Annotated C++ Reference Manual* was published by Bjarne Stroustrup. In this discussion, I will only talk about .NET Framework base classes and .NET Framework library classes with respect to objects, because that seems to be the main focus of where we are going today.

When Bjarne Stroustrup published the above book on C++ or C with classes, we started associating the word class and object with the term *abstract data type*. But what is the difference between data types and abstract data types? A data type is a set of values. Some algorithm then operates upon managing and changing the set of values. An abstract data type has not only a set of values, but also a set of operations that can be performed upon the set of values. The main idea behind the abstract data types is the separation of the use of the data type from its implementation. Figure 1.1 shows the four major parts of an abstract data type. Syntax and semantics define how an application program will use the abstract data type. Representation and algorithms show a possible implementation.

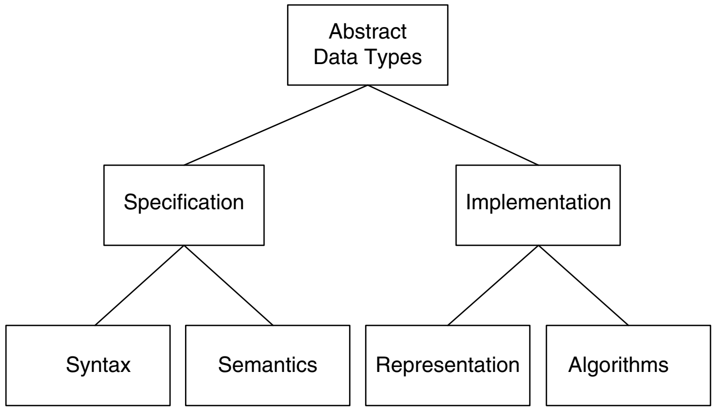


Figure 1.1 Abstract Data Type

For an abstract data type, we have therefore defined a set of behaviors, and a range of values that the abstract data type can assume. Using the data type does not involve knowing the implementation details. Representation is specified to define how values will be represented in memory. We call these representations *class member variables* in VB.NET or C#. The algorithm or programs specify how the operations are implemented. We call these programs *member functions* in VB.NET or C#. The semantics specify what results would be returned for any possible input value for each member function. The syntax specifies the VB.NET or C# operator symbols or function names, the number and types of all the operands, and the return values of the member functions. We are therefore creating our own data object (abstract data type) for the software to work with and use. This is opposed to only using the data types predefined by the compiler, such as integer, character, and so on. These abstract data types or objects, as defined in Grady Booch's book *Object-Oriented Analysis and Design with Applications, Third Edition* (2007), are as follows: "an object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain."

Another classic book relating to objects is *Design Patterns* (Gamma 1995). This books points out the elements of reusable object-oriented software.

1.3 Gaining an Understanding

We have slowly come to the realization of just what properties our program should have to make it work in solving complex real world problems. Having a new language like VB.NET or C# and their associated capabilities to create classes and objects was not enough. We realized that just using the abstract data type or class was not enough. As part of this ongoing development, the methodology called object-oriented technology evolved into what is called the object model. The software engineering foundation whose elements are collectively called the object model encompass the principles of abstraction, modularity, encapsulation, hierarchy, typing, concurrency, and persistence. The object model defines the use of these elements in such a way that they form a synergistic association.

As with any discipline, such as calculus in mathematics, we need a symbolism or notation in which to express the design of the objects. The creation of the C++ language, as an example, supplied one language notation needed to write our object-oriented programs. However, we still needed a notation for the design methodology to express our overall approach to the software development. In 1991, Grady Booch first published his book *Object-Oriented Analysis and Design with Applications* in which he defined a set of notations. These notations have become the *defacto* standard for Object Oriented Design. His second edition does an even better job of describing the overall Object Oriented Design notation and the object model. In this second edition, he expresses all examples in terms of the C++ language, which for a time became the predominate language for object-oriented software development. We even have a Windows GUI tool based upon this notation to aid us in our thinking. This tool by Rational Corporation and Grady Booch was called ROSE. Quite a change from how calculus and its notation were initially used. We almost immediately have the same engine we wish to program on, aiding us in doing the programming. This tool has continued to evolve and is now called the Universal Modeling Language (UML).

An object (or component) then is an entity based upon abstract data type theory, implemented as a class in a language such as VB.NET or C#, and the class incorporates the attributes of the object model. What we have been describing, however, is just the tip of the iceberg relative to objects. The description so far has described the static definitions and has not talked about objects talking with other objects. Let's just look at one of the object model attributes: inheritance. Inheritance is our software equivalent of the integrated electronic circuit (IC) manufacturing technique of large-scale integration (LSI) that allows such tremendous advances in electronic system creations. Software using inheritance is certainly very small scale at the present, but the direction is set. Inheritance allows the creating of a small-scale integration (SSI) black box in software. This SSI creates an encapsulated software cluster of objects directed toward the solution of some function needed for the application. We have thus abstracted away a large amount of the complexity and the programmer works only with the interfaces of the cluster. The programmer then sends messages between these clusters, just like the electronic logic designed has wires between ICs, over which signals are sent.

1.4 Software Components

Although we allude to software components having an analogy to hardware chips, this is only true in a most general sense. Software components created with the rich vocabularies of the programming language, and based upon the constructs created by the programmer's mind, have a far greater range of flexibility and power for problem solving than hardware chips. Of course, therein lays a great deal of the complexity nature of software programs. However, the software components ride on top of the hardware chips adding another complete level of abstraction. The deterministic logic involved in a complex LSI chip is very impressive. But the LSI chip is very limited in the possibility of forming any synergist relationship with a human mental object.

The more we dwell upon the direction of the .NET Framework's object model, in all its technologies, the more it seems to feel like we are externalizing the mind's use of mental object behavior mechanics. Certainly, the object relationships formed with linking and embedding of software objects, via interfaces, doesn't look much like the dendrite distribution of influences on clusters of neurons. But certainly now, one software object is starting to effect one or more other software objects to accomplish its goal.

Let's look at a control object or collection of control objects from an everyday practical standpoint that we are using in other engineering fields. One of our early loves is the automobile. We can hardly wait to learn how to drive one. Notice, we said drive one, any one. We have done such a great job on our encapsulation and interface exposure that we can learn to drive any kind and be able to drive any other kind. The automobile object we interact with has three primary interface controls: steering wheel, throttle, and brake. We realize that encapsulated within that automobile object is many internal functions. We can be assured that these control interfaces will not change from automobile object to automobile object. In other words, if we go from a General Motors car to a Ford car we can depend on the same functionality of these control interfaces.

Another characteristic of a software object is persistence. Persistence of an object is learned very early by a child. Eventually, when we show a child a toy and then hide it behind our back, the child knows the toy still exists. The child has now conceptualized the toy object as part of its mental set of objects. As the programmer does a mental conceptualization of various software objects, this will lead to a high level of persistence of the objects in the programmer's mind. Because one of the main features of standard software objects is reusability, the efficiency of the programmer will continue to increase as the standard objects are conceptualized in the programmer's mental model.

Polymorphic behavior is another characteristic that can be implemented in a software object. Probably one of the earlier forms that a child realizes has different behavior, based upon form, is the chair object. The chair object is polymorphic in that its behavior depends on its form. We have rocking chairs, kitchen chairs, lounge chairs, and so on. This idea of form and related behavior has created a whole field of study called morphology. Certainly, this is a key idea in how we relate cognitively to various objects. Not only does the clustering of our objects have form relationships, the internal constructs of the objects have a form relationship. There is a definite relationship between the logic flow of a program and the placement of the various meaningful chunks of a program. This is somewhat different than a pure polymorphic nature of a function, but does point out that we should be aware of the morphology of our objects and their parts and placement in our program.

INDEX

A

abstract data types
  data types vs., 8–9
  objects (components) based on, 10
ACLs (access control lists), in KMDF, 180
AcquireLock method, IWDFObject, 70
Active Template Library (ATL), UMDF, 85, 112
adaptive time-outs, I/O requests, 66–67
AddRef method, IUnknown
  defined, 82, 95
  implementing, 125
  rules for reference counting, 120
  UMDF object model, 45
Administrators, privileges of, 180
algorithms, 8–9
Algorithms + Data Structures = Programs (Wirth), 8
The Annotated C++ Reference Manual (Stroustrup), 8
applications, UMDF driver architecture, 44, 83
architecture
  device driver objects, 7–12
  UMDF, 82–84
  WDF. *See* WDF (Windows Driver Foundation)
  Windows I/O layered, 79–81
  WMI, 253–254
Aristotle, on characteristics of objects and concepts, 7
ATL (Active Template Library), UMDF, 85, 112
attributes
  initializing for KMDF objects, 220–221
  initializing for WDFDEVICE, 229–231
  of KMDF objects, 144–145
  setting, 237, 240–241
automatic forwarding, UMDF drivers configuring, 54–55
automobiles, functionality of control interfaces in, 11–12

B

base classes, WMI, 257–260
behaviors, defining abstract data type, 9
Brooch, Grady, 7
buffers
  creating for I/O requests, 63–64, 157–158
  performing buffered I/O, 301–302
  retrieving from I/O requests, 61, 155–156
build.exe, WDK build utility, 183
builds, KMDF
  building a project, 186–187
  building Toaster example, 187–190
  types of, 185–186
  WDK build tools, 183–185
builds, UMDF, 75–77
bus drivers, in KMDF
  creating, 29
  overview of, 135, 137–138

C

C programming language
  DDIs (device-driver interfaces), 129
  driver development and, 183
C# programming language, abstract data type behaviors in, 9
C++ How to Program, Seventh Edition (Deitel 2009), 112
C++ programming language
  development of, 8
  driver development and, 183
  object-oriented software development and, 10
  supplying notation for writing OOP, 10

C++ template libraries, UMDF drivers using, 112
callbacks
  code for registering, 245–248
  code for setting self-managed I/O, 315–316
  creating and deleting objects, 53–55, 147, 239–240
  device callback object, 100–106
  driver callback object, 96–100
  EvtInterruptDpc callback, 281–282
  EvtInterruptIsr callback, 279–281
  interfaces for UMDF driver, 47–49
  I/O request suspension, 65
  Plug and Play support and, 227
  power management notification, 71
  for power-up/power-down, 250–251
  registering for Plug and Play and power management, 243–245
  retrieving buffers from I/O requests, 62
  self-managed I/O, 67–68, 161
  surprise removal sequence, 75
  UMDF driver implementation and, 122–125
  wake signal support, 251–252
cancellation
  guidelines for I/O requests, 66–67
  integrated I/O queuing and, 22–26
  of I/O request, 64–65
CAs (certificate authorities), obtaining signed catalog file from, 194
catalog files (.cat)
  in KMDF driver package, 190
  obtaining for driver package, 193–194
certificate authorities (CAs), obtaining signed catalog file from, 194
checked builds, types of builds, 185
class factories, implementing UMDF drivers with, 122
class IDs (CLSIDs), used by COM, 112
class member variables, defining abstract data type representations as, 9
classes
  definition of, 8
  implementing objects as, 10
  Skeleton driver, 89–90
  troubleshooting WMI, 265
  WMI class name, 257–260
cleanup requests
  deleting objects, 147, 239–240
  device callback, 105
  driver callback, 48
  in filter driver, 55
  I/O requests in KMDF, 149–150
  UMDF drivers handling, 53–55
client-server model, COM based on, 82
close requests
  devices, 105
  files, 48, 235–237
  in filter driver, 55
  UMDF drivers handling, 53–55
CLSIDs (class IDs), used by COM, 112
CMyDriver object, Skeleton driver, 91
CMyDriver::CreateInstance method, 96–98
co-install DLL, WDF, 191
co-installer section, INF file, 76
COM (component object model)
  UMDF driver development and, 81–82
  UMDF interfaces based on, 45
  UMDF objects based on, 19–20, 45
  User Mode Driver host process, 44–45
COM (component object model), creating UMDF drivers
    basic infrastructure implementation, 120–126
    COM fundamentals and, 112–114
    getting started, 111–112
    HRESULT, 114–116
    overview of, 111
    using COM objects, 116–120
COM (component object model), UMDF support for device callback object, 100–106
  driver callback object, 96–100
  IClassFactory methods, 96
  IUnknown methods, 95
  overview of, 95
COM clients, 116
Complete or CompleteWithInformation method, IWDFIoRequest, 65–66
components
  defined, 7
  KMDF, 131–132
  software components, 11–12

WDF, 13–14
WMI exporting information from drivers to other components, 172–173
Comsup.cpp source file, 95, 108
Comsup.h source file, 95
concurrency
managing in Windows drivers, 22–23
UMDF and, 69
Configure method, device callback objects, 105–106
context. *See* object context
control device drivers, for legacy (NT 4.0-style) devices, 29
control device objects
in KMDF, 138–139
overview of, 135
in UMDF, 52
counted UNICODE strings, KMDF security and, 181
create requests
CreateRequest method, IWDFDevice, 62
driver callback, 47–48
files, 235–237
filter drivers, 55
flow of I/O control requests, 56–57
UMDF drivers handling, 53–55
CreateInstance method, IClassFactory
creating driver callback object, 96
defined, 82, 96
implementing, 122
UMDF driver functionality, 87
CreateInstance method, of device callback object, 101–102
CreateIOQueue method, IWdfDevice, 109
CreatePreallocatedWdfMemory method, IWdfDriver, 61, 63–64
CreateRequest method, IWDFDevice, 62
CreateWdfMemory method, IWdfDriver, 63–64
CUnknown class, 123
CUnknown method, IUnknown, 95

D

data providers, registering drivers as WMI data provider, 254–255
data types, abstract data types vs., 8–9
Data Types (Cleaveland), 8
DDIs (device-driver interfaces)
C-language, 129
driver frameworks, 28
in KMDF drivers, 183
in WDM drivers, 14–15
debugger extensions
for KMDF drivers, 204–205
for UMDF drivers, 75
for WDF drivers, 37
in WinDbg, 201
debugging
driver verification, 198
drivers, 198–200
kernel mode drivers, 200–201
macros and routines for, 203
PREfast tool for, 196–197
registry settings and, 201–203
SDV (Static Driver Verifier) tool, 197
symbols file and, 203
trace logging and, 198
WinDbg applied to Toaster example, 205–208
WinDbg commands, 200–201
WPP tracing and, 205
defaults
configuring default I/O queue, 232–233
execution level in KMDF, 177
safe defaults in KMDF security, 180
synchronization scope in KMDF, 238
design
WDF component functions for, 13
WDF goals, 14–15
Design Patterns (Gamma), 9
DestinationDirs section, INF file, 76
device callback object
overview of, 100–106
Skeleton driver, UMDF, 96–100
device driver architecture overview
objects, 7–12
WDF. *See* WDF (Windows Driver Foundation)
device I/O requests
code for creating queues for device I/O control requests, 297–298
code for finding manual requests, 304–308
code for handling, 299–301

device I/O requests (*contd.*)
KMDF, 150
UMDF drivers, 56–57
WDF architecture, 25–26
device objects (DOs), KMDF
bus drivers and PDOs, 135, 137–138
filter drivers and FDO, 135–136
function drivers and functional device objects, 135–137
legacy drivers and control device objects, 138–139
overview of, 135–136
power-down and removal of FDOs, 168–169
power-down and removal of PDOs, 169–170
startup sequence for FDOs, 165–166
startup sequence for PDOs, 166–167
device property store, UMDF, 50–51
device scope
KMDF synchronization, 175, 238
UMDF synchronization, 69–70
WDF architecture, 23
device usage model, KMD driver samples listed by, 213–215
Device.cpp, adding device-specific code to, 109
device-driver interfaces. *See* DDIs (device-driver interfaces)
devices
control device objects in. *See* control device objects
creating device interface, 231–232
enumeration and startup in KMDF, 164
enumeration and startup in UMDF, 71–72
filters. *See* filter DOs (filter device objects)
functional device objects. *See* FDOs (functional device objects)
initializing device context area, 229–231
KMDF supported, 129–131
physical device objects. *See* PDOs (physical device objects)
power-down and removal in KMDF, 168–170
power-down and removal in UMDF, 72–75
safe naming techniques, 181
setting attributes, 237, 240–241
startup sequence for, 165–167
support for device interrupts, 274
support in WDF, 15–16
supported in User Mode, 42
surprise removal sequence in KMDF, 170–172
DIFx (Driver Install Frameworks), 14
digital signatures, 193–194
direct memory access (DMA), 42
Dirs, optional files in builds, 184
dispatch execution level, KMDF synchronization, 178
dispatch types
handling WMI requests, 255–256
I/O queues, 58–59, 153–154
KMDF supported, 291–292
DLL_PROCESS_ATTACH, 91–92
DLL_PROCESS_DETACH, 91–92
DllGetClassObject function, UMDF
defined, 49
driver functionality, 84–86
implementing UMDF driver infrastructure, 121–122
overview of, 93–95
DllMain export
as driver entry main point, 91–92
implementing UMDF driver infrastructure, 121
UMDF driver functionality and, 84–86
DLLs (dynamic-link libraries)
in KMDF driver package, 190
UMDF as, 33, 82–83
UMDF driver functionality and, 84–85
WDF debugger extensions in, 37
WMI providers and, 253
DMA (direct memory access), 42
DMDF. *See* KMDF (Kernel Mode Driver Framework)
DOs. *See* device objects (DOs)
down device object, I/O requests, 52
DPC, deferred processing of interrupts, 281
driver callback object
creating, 98–100
implementing UMDF driver with class factory, 122
UMDF functions supporting COM, 96–100
driver frameworks, WDF, 28–32
driver host process, in Windows kernel, 32
driver information, web sites for, 323–330
Driver Install Frameworks (DIFx), 14

driver manager
in UMDF driver architecture, 43–44, 83–84
in Windows kernel, 33
driver model, KMDF, 129–131
driver model, UMDF, 43–45
driver model, WDF
functions of, 13–14
overview of, 16–17
support for Windows 7, 15–16
driver roles, KMDF, 135–136
driver signing, WDF, 14
driver.cpp file
adding device-specific code to, 109
creating driver callback object, 98
DriverEntry object
creating, 225–227
in KMDF drivers, 132
dynamic binding, in KMDF, 208
dynamic testing, in KMDF, 196
dynamic-link libraries. *See* DLLs (dynamic-link libraries)

E

Echo driver, UMDF, 42, 88
enumeration
KMDF devices, 164
UMDF devices, 71–72
ERROR_CANCELLED, I/O requests, 65
errors, detecting with PREfast utility, 34–35
ETW (Event Tracing for Windows)
defined, 14
Kernel Mode Drivers using, 36–37
UMDF drivers using, 75
events
drivers implementing callback interfaces for important, 47–49
driver's WMI events are not being received, 267
firing WMI events, 260–265
KMDF object model, 139–141
trace events, 269–271
WDF object model, 18
WMI system event log, 269
EvtCleanupCallback routine, 147, 239–240
EvtDestroyCallback routine, 147
EvtDevicePrepareHardware, 283
EvtDeviceReleaseHardware, 283
EvtDriverDeviceAdd callback
managing power policy, 248–249
Plug and Play support and, 227
registering callbacks, 243
EvtInterruptDisable callback, 277
EvtInterruptDpc callback, 281–282
EvtInterruptEnable callback, 276–277
EvtInterruptlsr callback, 279–281
EvtIo* callback, 132
execution levels, KMDF
interaction with synchronization scope, 179–180
overview of, 177–178
setting, 239
Exports.def file, 107
extensions. *See* debugger extensions

F

Facility field, HRESULT, 114–115
FAILED macro, HRESULT, 115–117
FDOs (functional device objects)
overview of, 135–137
power-down and removal, 168–169, 250
startup sequence for, 165–166
features
KMDF driver, 215–216
UMDF driver, 49–51
fields, HRESULT, 114–115
file close request, I/O requests in KMDF, 235–237
file create request, I/O requests in KMDF, 235–237
filter DOs (filter device objects)
overview of, 135–136
power managed queues and, 152–153
filter drivers
create, cleanup, and close in, 55
KMDF supporting creation of, 29
UMDF drivers identifying themselves as, 55
flags, WDK, 186
frameworks, WDF
component functions for, 13–14
verifier, 36
free builds, types of builds, 185

functional device objects. *See* FDOs (functional device objects)
function drivers
KMDF, 29, 135–137
UMDF, 55
FxDeviceInit, 102–105
FxWdfDriver, 118

G
GetDefaultIoTarget method, IWDFDevice, 61–62
globally unique identifiers. *See* GUIDs (globally unique identifiers)
GUIDs (globally unique identifiers)
guidgen.exe, 258
used by COM, 112–113
WMI and, 172

H
HAL (Hardware Abstraction Layer), 2
hardware resources
code for mapping, 284–288
code for unmapping, 288–289
mapping, 283–284
hierarchical arrangement
of KMDF objects, 141–144
of objects, 18
host process, UMDF driver architecture, 43–45, 82–84
HRESULT
overview of, 114–116
testing for simple success or failure, 117

I
IClassFactory interface, UMDF
defined, 82
driver support for, 49, 85–87
implementing UMDF driver infrastructure, 122
methods, 96
IDeviceInitialize interface, 109
IDriverEntry interface, UMDF
device enumeration and startup, 72
driver support for, 48–50, 85–87
methods, 98
Skeleton driver, 90–91
IDWFxx interfacesUMDF, 46–47
IFileCallbackCleanup interface, UMDF
defined, 48
device callback object, 105
handling cleanup and close in function drivers, 55
handling cleanup request, 53–55
IFileCallbackClose interface, UMDF
defined, 48
device callback object, 105
handling cleanup and close in function drivers, 55
handling cleanup request, 53–55
IFR (in-flight recorder), 36
IIDs (interface IDs)
implementing QueryInterface, 125
used by COM, 112–113
IImpersonateCallback interface, UMDF, 50
Impersonate method, IWDFIoRequest, 50
impersonation, UMDF driver, 50
INF files
creating device interface, 231
for KMDF drivers, 190–193
specifying maximum impersonation level of UMDF driver, 50
for UMDF drivers, 76–77
using Skeleton driver as basis for development, 108
for WDF drivers, 14
in-flight recorder (IFR), 36
in-flight requests, suspending, 160
inheritance, software using, 10–11
Initialize method
adding device-specific code to Device.cpp, 109
device callback object, 102–105
installation
KMDF drivers, 190–193
in Toaster example, 194–196
UMDF drivers, 76–77
WDF drivers, 14
integrated I/O queuing and cancellation, WDF, 22–26, 117
interface IDs (IIDs)
implementing QueryInterface, 125
used by COM, 112–113

interface pointers, COM
obtaining interface on UMDF object with, 117–118
overview of, 113–114
reference counting and, 120
interfaces
COM, 82, 113
creating device interface for Simple Toaster example, 231–232
obtaining on UMDF objects, 117–119
Skeleton driver, 90–91
UMDF, 45–47
ways to create device interfaces, 231–232
internal device I/O request, KMDF, 150
internal trace logging, 36–37
interrupt request level (IRLQ), 23
interrupts
code for creating, 275–276
code for enabling, 276–277
code for EvtInterruptDpc callback, 281–282
creating, 274–275
deferred processing of, 281
enabling/disabling, 276
handling, 278–279
overview of, 273
post- interrupt enable and pre-interrupt disable processing, 277–278
support for, 274
writing Kernel Mode Drivers for handling, 42
InterruptService routines, 278
INX files
optional files in builds, 184
using Skeleton driver as basis for development, 108
I/O manager
defined, 1–2
in Windows kernel, 32–33
I/O mapped resources, 283–284
I/O model, KMDF
accessing IRPs and WDM structures, 161–162
cancelling/suspending requests, 158–160
completing requests, 160–161
creating buffers for requests, 157–158
integrated queuing and cancellation, 22–26, 111
overview of, 147–149
queues, 152–154
request handler, 149–151
request objects, 154
retrieving buffers from requests, 155–156
self-managed callbacks, 161
targets, 156–157
Windows layered architecture for, 79–81
I/O queues
adding device-specific code to Device.cpp, 109
configuring, 293–294
integrated I/O queuing and cancellation, 22–26, 47–49
programming. *See* programming I/O queues
I/O queues, KMDF
configuring, 222–223
dispatch types and, 153–154
power management and, 152–153
WDFQUEUE object, 152
I/O queues, UMDF
callback interfaces, 47
dispatch types, 58–59
dispatching I/O request to UMDF driver, 53
overview of, 56–58
power management and, 59–60
I/O queues, WDF
integrated queuing and cancellation, 22–26, 279–281
interfaces for UMDF object types, 20
I/O request packets. *See* IRPs (I/O request packets)
I/O requests
code for finding, 304–308
code for handling device I/O requests, 299–301
forwarding requests to queues, 302–303
retrieving requests from manual queues, 303–304
WMI, 255–256
I/O requests, KMDF
cancelling/suspending, 158–160
completing, 160–161
configuring, 224
creating, cleaning up, and closing, 149–150
file create and close requests, 235–237
flow of requests through request handler, 151
handling in Simple Toaster example, 233–235
reading, writing, device I/O control, and internal device I/O control, 150

I/O requests, KMDF (*contd.*)
request handler, 149–151
request objects, 154
retrieving buffers from, 155–156
I/O requests, UMDF
adaptive time-outs, 66–67
canceled and suspended, 64–66
completing, 66
creating buffers for, 63–64
I/O queues and, 56–60
overview of, 51–56, 60–61
retrieving buffers from, 61
self-managed I/O, 67–68
sending to I/O target, 61–63
I/O requests, WDF
driver frameworks, 28–30
driver model, 16
integrated queuing and cancellation, 22–26, 232–233
overview of, 24–25
I/O target
creating buffers for I/O requests, 63–64
in KMDF, 156–157
sending I/O requests to, 61–63
IObjectCleanup interface, 48
IOCTL requests, parameter validation and, 180–181
IPnpCallback interface, UMDF
adding device-specific code to Device.cpp, 109
driver callback, 47–48
power management notification, 71
surprise removal sequence, 74–75
IPnpCallbackHardware interface, UMDF
driver callback, 47–48
power management notification, 71
surprise removal sequence, 75
IPnpCallbackSelfManagedIo interface, UMDF
driver callback, 47–48
self-managed I/O callbacks, 67–68
IQueueCallbackCreate interface
create, cleanup, and close in filter drivers, 55
create request in function drivers, 55
driver callback, 47–48
handling create requests, 53–55
IQueueCallbackDefaultIoHandler interface, 48
IQueueCallbackDeviceIoControl interface, 49
IQueueCallbackIoResume interface, 49
IQueueCallbackIoStop interface, UMDF
defined, 49
I/O request suspension, 65
power-managed queues, 59
IQueueCallbackRead interface, UMDF, 47–49
IQueueCallbackWrite interface, UMDF, 49
IRequestCallbackCancel interface, UMDF
defined, 49
I/O request cancellation, 65
retrieving buffers from I/O requests, 62
IRequestCallbackCompletion interface, UMDF, 62
IRequestCallbackRequestCompletion interface, UMDF, 49
IRLQ (interrupt request level), 23
IRP_MJ_SYSTEM_CONTROL requests, 172–173, 255–256
IRPs (I/O request packets)
accessing from KMDF, 161–162
creating in Windows kernel, 32–33
handling WMI requests, 255–256
I/O request flow to UMDF driver and, 52–53
overview of, 23–24
Windows I/O architecture and, 81
WMI IRPs and system event log, 269
IsEqualID function, comparing IIDs, 125–126
IUnknown interface, COM
as core COM interface, 112
defined, 82
device callback objects and, 105
implementing UMDF callback objects, 123–125
method names, 95
IWDFDevice interface, UMDF
creating targets, 62
device callback object, 105
self-managed I/O callbacks, 67–68
sending I/O requests to I/O target, 61–62
Skeleton driver, 91
IWDFDeviceInitialize interface, UMDF
configuring automatic forwarding, 54–55
create, cleanup, and close in filter drivers, 55
device callback object and, 102–105
driver callback object and, 98
Skeleton driver, 91
UMDF driver creating property store, 50–51

IWdfDriver interface, UMDF
creating buffers for I/O requests, 63–64
creating driver callback object, 98
retrieving buffers from I/O requests, 61
Skeleton driver, 91
IWDFFileHandleTargetFactory interface, 62
IWdfIoQueue interface, UMDF
device callback object, 105
overview of, 56–58
Start method, 60
Stop method, 60
StopSynchronously method, 60
IWDFIoRequest interface, UMDF
impersonation requests, 50
I/O request cancellation, 65
UMDF driver impersonation requests, 66
IWDFIoTarget interface, UMDF, 61–62
IWDFIoTargetStateManagement interface, UMDF, 63
IWDFloRequest interface, UMDF, 60–61
IWDFMemory interface, UMDF, 61
IWDFNamedPropertyStore interface, UMDF, 50–51
IWDFObject interface, drivers, 70
IWDFoRequest interface, UMDF, 62
IWDFUsbTargetFactory interface, UMDF, 62

K

Kant, Immanuel, 7
KD, for kernel debugging, 199
Kernel Mode Driver Framework. *See* KMDF (Kernel Mode Driver Framework)
kernel mode drivers
debugging, 200–201
internal trace logging for, 36–37
UMDF driver architecture and, 43, 45, 82–84
WDF component functions for, 13–14
WMI requests for, 27
writing, 42
KMDF (Kernel Mode Driver Framework)
bus drivers and physical device objects, 135, 137–138
comparing KMDF drivers with WDM drivers, 132–135
components of, 131–132
design goals for WDF, 15
device and driver support in WDF, 15–16
device objects and driver roles, 135–136
driver structure, 132
filter drivers and filter device objects, 135–136
function drivers and functional device objects, 135–137
internal trace logging for, 36–37
I/O model. *See* I/O model, KMDF
legacy drivers and control device objects, 138–139
object model. *See* object model, KMDF
overview of, 129
plug and play and power management support. *See* Plug and Play and power management
programming drivers for. *See* programming KMDF drivers
programming hardware drivers. *See* programming KMDF hardware drivers
supported devices and driver types, 129–131
understanding, 28–30
WDF component functions for, 14
WDF driver model, 16–17
KMDF installation and build
building a project, 186–187
building Toaster example, 187–190
catalog files and digital signatures, 193–194
debugger extensions, 204–205
debugging drivers, 198–200
debugging kernel mode drivers, 200–201
driver verification, 198
installing drivers, 190–193
installing Toaster example, 194–196
macros and routines for debugging, 203
overview of, 183
PREfast debugging tool, 196–197
registry settings and debugging features, 201–203
SDV (Static Driver Verifier) tool, 197
symbols file and debugging features, 203
testing approaches, 196
trace logging, 198
types of builds, 185–186
versioning and dynamic binding and, 208–209
WDK build tools, 183–185
WinDbg applied to Toaster example, 205–208
WPP tracing and, 205
KMDF Verifier, 198

L
legacy drivers, in KMDF, 138–139
lines of code, comparing KMDF drivers with WDM drivers, 134
LocalService security context, UMDF drivers, 50
locking constraint, UMDF, 23, 69
locks
  comparing KMDF drivers with WDM drivers, 134
  KMDF drivers, 178–179
  UMDF drivers, 70
LockServer method, IClassFactory, 96, 122
logs
  system event log, 269
  trace logging. *See* trace logging
  WMI WDM provider log, 269

M
macros
  for debugging in KMDF, 203
  for declaring object context, 221–222
  initializing context area and attributes for device objects, 230–231
  used in KMDF samples, 218–219
  using HRESULT, 115–117
makefile
  required files in builds, 184
  in Toaster sample, 187–188
Makefile.inc
  optional files in builds, 184
  in Toaster sample, 187–188
managed object format (.mof) resource files, 184
manual dispatch type
  code for finding manual requests, 304–308
  creating manual queues, 294
  I/O queues and, 59
  KMDF and, 154, 291–292
  retrieving requests from manual queues, 303–304
mapping hardware resources
  code for, 284–288
  overview of, 283–284
MarkCancelable method, IWDFIoRequest, 65
member functions, abstract data type algorithms as, 9
memory
  creating buffers for I/O requests, 63–64
  retrieving buffers from I/O requests, 61
memory-mapped resources, 284
methods
  COM, 113
  KMDF objects, 139–141
  return from COM, 114–116
  unable to access driver's WMI method, 266
  WDF objects, 18
.mof (managed object format) resource files, 184
morphology, of objects, 12

N
names
  KMD driver samples listed by, 211–212
  security of KMDF names, 181
  simplifying GUID, 112–113
network-connected devices, UMDF support for, 42
no scope
  KMDF, 175, 238
  UMDF, 70
  WDF, 23
nonpaged pools, writing kernel mode drivers for, 42
nonpower-managed queues, UMDF, 60
notations, Object Oriented Design, 10
NT_SUCCESS macro, 115
NTSTATUS, converting into HRESULT, 116–117
NTTARGETFILES statement, 107

O
OBG (Object Management Group), 8
object context
  initializing device context area, 229–231
  in KMDF object model, 145–146
  programming KMDF drivers and, 221–222
Object Management Group (OBG), 8
object model
  classes incorporating attributes of, 10
  defined, 10
  inheritance as attribute of, 10–11
  software components, 11–12
  UMDF, 45
  WDF, 17–20

object model, KMDF
creating objects, 146–147, 220–221
deleting objects, 146–147, 239–240
driver structure and concepts and, 219
hierarchical structure of, 141–144
methods, properties, and events, 139–141
object attributes, 144–145
object context, 145–146, 221–222
object types, 18–19
overview of, 139
setting object attributes, 144–145, 237, 240–241
types of objects, 142–144
Object-Oriented Analysis and Design with Applications, Third Edition (Booch), 9, 10
Object Oriented Programming (OOP)
evolution into object model, 10
software techniques, 7
objects
COM, 112–113
creating driver objects, 225–227
defining software, 8–9
driver callback, 47–49
nature of, 7–8
Skeleton driver, 90–91
software components, 11–12
UMDF driver, 45–47
understanding, 10–11
OnCancel method, IRequestCallbackCancel, 62, 65
OnCompletion method, IRequestCallbackCompletion, 62
OnCreateFile method, IQueueCallbackCreate, UMDF create request in function drivers, 55
OnDeInitialize method, IDriverEntry, 85, 98
OnDeviceAdd method, IDriverEntry
adding device-specific code to Driver.cpp, 109
creating driver callback object, 98–100
device enumeration and startup, 72
UMDF driver functionality, 85–87
OnImpersonation method, UMDF drivers, 50
OnInitialize method, IDriverEntry, 85–87, 98
OnIoStop method, IQueueCallbackIoStop, 59, 65
OnReleaseHardware method, IPnpCallbackHardware, 75
OnSurpriseRemoval method, IPnpCallback, 47, 74–75
OOP (Object Oriented Programming)
evolution into object model, 10
software techniques, 7
Operation, in KMDF object model, 139

P

PAGED_CODE macro, 218
parallel dispatch type
creating parallel queues, 294
handling requests from parallel queues, 298–299
I/O queues and, 58
KMDF supported, 154, 291–292
parameter validation, KMDF security, 180–181
parent/child relationships, in KMDF object model, 141–144
passive execution level, KMDF synchronization, 177
PCIDRV driver. *See* also programming KMDF hardware drivers, 273
PDOs (physical device objects)
in KMDF, 137–138
overview of, 135
power-down and removal, 169–170
power managed queues and, 152–153
power-up/power-down, 251
startup sequence for, 166–167
performance goals, impacting device driver authors, 2–3
persistence, implementing in software object, 12
PFD (PREFAST for Drivers). *See* PREfast debugging tool
physical device objects. *See* PDOs (physical device objects)
Plato, on characteristics of objects and concepts, 7
Plug and Play and power management
flow of I/O control requests, 56–57
integrated I/O queuing and cancellation, 22–26, 55
I/O queues and, 59–60
overview of, 20–22
programming. *See* Programming Plug and Play and power management

Plug and Play and power management (*contd.*)
UMDF notification, 70–71
UMDF support for, 31
WDF driver model including, 16
WDF support for, 20–22
Plug and Play and power management, in KMDF
counted UNICODE strings, 181
device enumeration and startup, 164
execution levels, 177–178
interactions of synchronization mechanisms, 179–180
I/O queues and, 152–153
locks, 178–179
overview of, 163–164
parameter validation, 180–181
power down and removal of filter DOs, 168–169
power down and removal of physical DOs, 169–170
safe defaults, 180
safe device naming techniques, 181
security, 180
startup sequence for function and filter DOs, 165–166
startup sequence for physical DOs, 166–167
support for, 29–30
surprise removal sequence, 170–172
synchronization issues, 173–175
synchronization scope, 175–177
WMI request handler, 172–173
Plug and Play callback interface, 47–48
policies
power management and, 244–245, 248–249
sample code for setting power policy, 249–250
polymorphic behavior, implementing in software object, 12
power down
callbacks for, 250–251
of filter DOs, 168–169
overview of, 167
of physical DOs, 169–170
self-managed I/O during device power down and removal, 314
surprise removal sequence, 170–172
UMDF device, 72–74
power management. *See* Plug and Play and power management
power up, callbacks for, 250–251
power-managed queues, UMDF, 59
PREfast debugging tool
source code analysis with, 196–197
UMDF drivers using, 76
WDF testing tool, 34–35
privileges, administrators, 180
programming I/O queues
buffered I/O and, 301–302
code for creating and initializing watchdog timer, 316–317
code for creating queues for device I/O control requests, 297–298
code for creating queues for read requests, 296–297
code for creating queues for write requests, 294–296
code for deleting watchdog timer, 320–321
code for finding requests, 304–308
code for handling device I/O requests, 299–301
code for reading/writing the registry, 309–312
code for restarting watchdog timer, 319–320
code for setting self-managed I/O callbacks, 315–316
code for starting watchdog timer, 317
code for stopping watchdog timer, 318
creating and configuring, 293–294
forwarding requests to queues, 302–303
handling requests from parallel queues, 298–299
implementing watchdog timers, 315
overview of, 291–293
reading/writing the registry, 308–309
retrieving requests from manual queues, 303–304
self-managed I/O device startup and restart, 313–314
self-managed I/O during device power down and removal, 314
watchdog timer for self-managed I/O, 312–313
programming KMDF drivers
creating objects, 220–221

driver structure and concepts and, 219
I/O queues, 222–223
I/O requests, 224
macros used in KMDF samples, 218–219
object context areas, 221–222
overview of, 211
samples listed by device usage model, 213–215
samples listed by features supported, 215–216
samples listed by name, 211–212
WDM samples compared with KMDF samples, 216–218
programming KMDF drivers, in Featured Toaster example
deleting objects, 239–240
file create and close requests, 235–237
overview of, 235
setting device object attributes, 237, 240–241
setting execution levels, 239
setting synchronization scope, 238
programming KMDF drivers, in Simple Toaster example
configuring default I/O queue, 232–233
creating device interface, 231–232
creating DriverEntry object, 225–227
initializing device context area, 229–231
I/O request handler, 233–235
overview of, 224
programming KMDF hardware drivers
code for creating interrupts, 275–276
code for enabling interrupts, 276–277
code for EvtInterruptDpc callback, 281–282
code for EvtInterruptIsr callback, 279–281
code for mapping resources, 284–288
code for unmapping resources, 288–289
creating interrupts, 274–275
deferred processing for interrupts, 281
enabling/disabling interrupts, 276
handling interrupts, 278–279
mapping resources, 283–284
overview of, 273
post-interrupt enable and pre-interrupt disable processing, 277–278
supporting device interrupts, 274
programming Plug and Play and power management
callback for wake signal support, 251–252
callbacks for power-up/power-down, 250–251
managing power policy, 248–249
overview of, 243
registering callbacks, 243–245
sample code for callbacks, 245–248
sample code for setting power policy, 249–250
programming UMDF drivers
brief COM information, 81–82
driver DDL and exports, 91–95
functions for COM support. *See* COM (component object model), UMDF support for
overview of, 79
required driver functionality, 84–87
sample drivers, 87–91
UMDF architecture, 82–84
using Skeleton driver as basis for development, 106–110
Windows I/O, 79–81
programming WMI support
class names and base classes, 257–260
event tracing, 269–271
firing events, 260–265
handling WMI requests, 255–256
overview of, 253
registering driver as WMI data provider, 254–255
requirements for WDM drivers, 256–257
testing driver support, 268–269
troubleshooting, 265–268
WMI architecture and, 253–254
properties
KMDF object model, 139–141
unable to access driver's WMI properties, 266
WDF object model, 18

Q

QI (query-interface), COM, 82
QueryIClassFactory method, IClassFactory, 96
query-interface (QI), COM, 82
QueryInterface method, IClassFactory, 96
QueryInterface method, IUnknown
defined, 82, 95
device callback object, 106

QueryInterface method, IUnknown (*contd.*)
implementing, 125–126
obtaining interface on UMDF object, 117, 119
obtaining interface pointer in COM, 113
UMDF object model, 45
QueryIUnknown method, IUnknown
defined, 95
device callback object, 105
queue scope
KMDF, 175, 238
WDF, 23
queues, I/O. *See* I/O queues

R
.rc (resource files), optional files in builds, 184
read requests
code for creating queues for, 296–297
creating manual queues, 294
flow of I/O control requests, 56–57
KMDF I/O requests, 150
reference counting
implementing UMDF callback objects, 125
overview of, 119–120
reflector
I/O request flow to UMDF driver, 52, 83–84
in UMDF driver architecture, 43–44
in Windows kernel, 32
registry
code for reading/writing, 309–312
KMDF debugging features and, 201–203
reading/writing, 308–309
Release method, IUnknown
defined, 82, 95
implementing, 125
rules for reference counting, 120
UMDF object model, 45
ReleaseLock method, IWDFObject, 70
removal, KMDF devices
filter DOs, 168–169
overview of, 167
physical DOs, 169–170
surprise removal sequence, 170–172
removal, of devices, 314
removal, UMDF device
overview of, 72–74
surprise removal sequence, 74–75
representations, abstract data type, 8–9
requests. *See* I/O requests
resource files (.rc), optional files in builds, 184
RetrieveDevicePropertyStore method, IWDFDeviceInitialize, 50–51
Return code field, HRESULT, 114–115
\root\wmi class, 265–266
ROSE tool, 10
routines, for debugging in KMDF, 203
run-time environment
UMDF driver architecture, 43
using UMDF COM objects in, 116–120

S
S_FALSE return value, HRESULT, 115–116
S_OK return value, HRESULT, 55, 115–116
SDDL (security description definition language), 181
SDV (Static Driver Verifier) tool
compile-time unit-testing with, 197
overview of, 35–36
PFD helping to prepare for, 35
security, KMDF
counted UNICODE strings, 181
overview of, 180
parameter validation, 180–181
safe defaults, 180
safe device naming techniques, 181
security, WMI, 267
security description definition language (SDDL), 181
self-managed I/O
callbacks in KMDF, 161
callbacks in UMDF, 67–68
code for creating and initializing watchdog timer, 316–317
code for deleting watchdog timer, 320–321
code for restarting watchdog timer, 319–320
code for setting self-managed I/O callbacks, 315–316
code for starting watchdog timer, 317
code for stopping watchdog timer, 318
during device power-down and removal, 314
device startup and restart, 313–314
implementing watchdog timers, 315
watchdog timer for, 312–313

semantics, abstract data type, 8–9
sequential dispatch type
  I/O queues and, 58
  KMDF support for, 154, 291–292
service control manager, 1–2
Service Processes, 1–2
serviceability, WDF support for, 37–38
SetFilter method
  IDeviceInitialize, 109
  IWDFDeviceInitialize, 55
SetLockingConstraint method, IWdfDeviceInitialize, 105
Severity field, HRESULT, 114–115
Skeleton driver, UMDF
  as basis for development, 106–110
  classes, objects and interfaces, 89–90
  component source files, 89
  defined, 42, 88
  device callback object, 96–106
  overview of, 88
small-scale integration (SSI) in software, 11
software
  defining objects, 8–9
  gaining understanding, 10–11
  on nature of objects, 7–8
software-only drivers, UMDF supporting, 42
source code files
  analysis with PREfast tool, 196–197
  required files in builds, 184
sources files
  required files in builds, 184
  in Toaster sample, 188–189
  using Skeleton driver as basis for development, 107
SOURCES statement, customizing Sources file, 107
spin locks, in KMDF synchronization, 179–180
SSI (small-scale integration) in software, 11
Standard Template Library (STL), UMDF drivers, 112
Start method, IWDFIoQueue, 60
startup
  for functions and filter DOs, 165–166
  KMDF devices, 164
  for physical DOs, 166–167
  self-managed I/O for, 313–314
  UMDF devices, 71–72
state machine, 21–22
state variables, 134
static analysis, 14
Static Driver Verifier. *See* SDV (Static Driver Verifier)
static testing, KMDF approaches to testing, 196
STL (Standard Template Library), UMDF drivers, 112
Stop method, IWDFIoQueue, 60
StopSynchronously method, IWDFIoQueue, 60
strict timing loops, kernel mode drivers for, 42
strings, preventing string handling errors in KMDF, 181
SUCCEEDED macro, HRESULT, 115–117
Support Processes, 1–2
surprise removal sequence
  KMDF devices, 170–172
  UMDF devices, 74–75
suspension, of I/O requests, 65–67
symbols file, KMDF debugging and, 203
synchronization, KMDF
  comparing KMDF drivers with WDM drivers, 134
  execution levels, 177–178, 239
  interactions of synchronization mechanisms, 179–180
  issues, 173–175
  locks, 178–179
  scope, 175–177
  scope options, 238
  setting synchronization scope for Featured Toaster example, 238
synchronization, UMDF
  issues, 68–70
  overview of, 69
synchronization scope
  KMDF and, 175–177
  options for, 238
  WDF and, 23
  syntax, abstract data type, 8–9
  system event log, 269

T

TARGETNAME statement, customizing sources file, 107
targets, I/O
  creating buffers for I/O requests, 63–64
  KMDF, 156–157
  sending I/O requests to, 61–63
template libraries, UMDF drivers using C++, 112
testing. *See also* debugging
  KMDF approaches to, 196
  UMDF drivers, 75–77
  WDF component functions for, 14
  WDF tools for, 33–38
  WMI driver support, 268–269
timers. *See* watchdog timer
Toaster sample
  building, 190
  installing, 194–196
  makefile and Makefile.inc, 187–188
  sources files in, 188–189
  WinDbg applied to, 205–208
trace logging
  KMDF, 198
  WDF component functions for, 14
  WDF support for, 36–37
  WMI, 269–271
  WPP, 205
troubleshooting WMI
  changes in WMI security settings not taking effects, 267
  driver's WMI events are not being received, 267
  overview of, 265–268
  unable to access driver's WMI properties and methods, 266

U

UMDF (User Mode Driver Framework)
  build, test and debug, 75–77
  COM support. *See* COM (component object model), UMDF support for
  design goals for WDF, 15
  developing drivers with COM. *See* COM (component object model), creating UMDF drivers
  device and driver support in WDF, 15–16
  device enumeration and startup, 71–72
  device power-down and removal, 72–75
  devices supported, 42
  driver callback interfaces, 47–49
  driver features, 49–51
  driver model, 43–45
  integrated I/O queuing and cancellation, 22–26, 179–180
  interfaces for object types, 19–20
  internal trace logging for, 36–37
  I/O queues, 56–60
  I/O request flow, 51–56
  I/O request objects, 60–67
  KMDF vs., 41
  locking constraint, 23
  locks, 70
  object model, 45
  objects, 45–47
  overview of, 28, 41
  Plug and Play and power management notification, 70–71
  Plug and Play and power management support, 20–22
  programming drivers for. *See* programming UMDF drivers
  self-managed I/O, 67–68
  synchronization issues, 68–70
  understanding, 31–32
  WDF component functions for, 14
  WDF driver model, 16–17
  when to use User Mode Drivers, 129
  in Windows kernel, 32–33
UML (Universal Modeling Language), 10
UNICODE, counted UNICODE strings, 181
unit testing, with SDV (Static Driver Verifier), 197
Universal Modeling Language (UML), 10
Universal Serial Bus (USB) devices, 42
UnmarkCancelable method, IWDFIoRequest, 65
UNREFERENCED_PARAMETER macro, 218–219
up device object, 52
updates, UMDF drivers, 77
USB (Universal Serial Bus) devices, 42
USB/Echo Driver, UMDF, 42
USB/Filter driver, UMDF, 42, 88

USB/FX2_Driver, UMDF, 42
User Mode DDI, 33
User Mode Driver Framework. *See* UMDF (User Mode Driver Framework)
user mode drivers
host process, 44–45
when to use User Mode Drivers, 129
uuidgen.exe, 258

V

values, of data types vs. abstract data types, 8
VB.NET programming language, 9
verification, using Static Driver Verifier for, 35–36
Verifier, KMDF, 198
versioning
KMDF drivers and, 208–209
UMDF drivers and, 77
WDF driver model including, 16
WDF support for, 14, 37–38
VTable pointers, COM, 114

W

wait locks, in KMDF synchronization, 179
wake signal support, 251–252
watchdog timer
code for deleting watchdog timer, 320–321
code for restarting watchdog timer, 319–320
code for setting self-managed I/O callbacks, 315–316
code for starting watchdog timer, 317
code for stopping watchdog timer, 318
implementing watchdog timers, 315
for self-managed I/O, 312–313
self-managed I/O device startup and restart, 313–314
self-managed I/O during device power down and removal, 314
Wbemtest, testing WMI support, 268
WDF (Windows Driver Foundation)
co-install DLL, 191
component functions, 13–14
defined, 3
design goals for, 14–15
device and driver support in, 15–16
driver frameworks, 28–32
driver model, 16–17
execution levels, 177–178
installation package, 190
integrated I/O queuing and cancellation, 22–26, 88
KMD driver samples listed by device usage, 213–214
KMD driver samples listed by features supported, 215–216
KMD driver samples listed by name, 211–212
locks, 178–179
object model, 17–20
obtaining interface on UMDF object, 117–118
Plug and Play and power management support, 20–22, 163
synchronization scope, 175–177
Windows kernel, 32–33
WMI requests for kernel mode drivers only, 27
WDF (Windows Driver Foundation), development and testing tools
debugger extensions, 37
defined, 13
frameworks verifier, 36
overview of, 33–34
PREfast debugging tool, 34–35
serviceability and versioning, 37–38
Static Driver Verifier (SDV), 35–36
trace logging, 36–37
Wdf section, INF file, 76
WDF_IO_QUEUE_CONFIG, 223
WdfDefault, 54–55
WDFDEVICE
creating and initializing, 227, 316–317
initializing context area and attributes for, 229–231
KMDF driver structure and concepts, 219
WDFDRIVER
creating, 226
KMDF driver structure and concepts, 219
as root object of KMDF object model, 141
WDFINTERRUPT, 274
WDFIOTARGET object, 156–157
WdfKd.dll, 37
WDFMEMORY object, 155–158
WDFQUEUE object, 152, 219
WDFREQUEST object, 154, 224
WdfTrue, 54–55

WDFWMIINSTANCE, 173
WDK (Windows Driver Kit)
building UMDF drivers with, 75
flags, 186
functions, 13–14
KMDF distributed via, 131
PFD tool in, 34–35
WDM (Windows Driver Model)
accessing WDM structures from KMDF, 161–162
comparing KMDF drivers with WDM drivers, 132–135
complexity and limitations of, 14–15
defined, 3
KMDF and, 183
KMDF as replacement for, 129–130
power state changes and, 163
signing WDF drivers same way as drivers in, 14
WDM samples compared with KMDF samples, 216–218
WMI requirements for WDM drivers, 256–257
WMI WDM provider log, 269
web sites, for driver information, 323–330
WinDbg
applied to Toaster example, 205–208
kernel debugging, 199–200
symbols files and, 203
types of commands, 200–201
WindDbg, 199
Windows Driver Foundation. *See* WDF (Windows Driver Foundation)
Windows Driver Kit. *See* WDK (Windows Driver Kit)
Windows Driver Model. *See* WDM (Windows Driver Model)
Windows Internals, Fifth Edition (Russinovich and Solomon), 79
Windows I/O architecture, 79–81
Windows kernel
UMDF, 43–44, 83
WDF, 32–33
Windows Management Instrumentation. *See* WMI (Windows Management Instrumentation)
Windows Software Trace Preprocessor. *See* WPP (Windows Software Trace Preprocessor)
WMI (Windows Management Instrumentation)
architecture data flow, 54–55, 253–254
class names and base classes, 257–260
event tracing, 269–271
exporting information from drivers to other components, 172–173
firing events, 260–265
for Kernel Mode Drivers only, 27
programming support for. *See* programming WMI support
registering driver as WMI data provider, 254–255
request handling, 255–256
requirements for WDM drivers, 256–257
testing driver support, 268–269
troubleshooting, 265–268
Wmic, testing WMI support in driver, 268
WmiEvent class, 258, 260
Wmimofck, testing WMI support in driver, 268
wmiprov.log, 269
WPP (Windows Software Trace Preprocessor)
applying to KMDF drivers, 205
kernel mode drivers using, 36–37
KMDF trace logging based on, 198
write requests
code for creating queues for, 294–296
creating parallel queues, 294
flow of I/O control requests, 56–57
KMDF I/O requests, 150
options for, 234
wudfdd.h, 123
WudfExt.dll, 37

Addison Wesley

THIS PRODUCT

informit.com/register

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams
SAFARI BOOKS ONLINE

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS | Safari Books Online

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters. Visit **informit.com/newsletters**.
- Access FREE podcasts from experts at **informit.com/podcasts**.
- Read the latest author articles and sample chapters at **informit.com/articles**.
- Access thousands of books and videos in the Safari Books Online digital library at **safari.informit.com**.
- Get tips from expert blogs at **informit.com/blogs**.

Visit **informit.com/learn** to discover all the ways you can access the hottest technology content.

Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit **informit.com/socialconnect**.

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE PEARSON

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS | Safari Books Online

Try Safari Books Online FREE

Get online access to 5,000+ Books and Videos

Safari Books Online

FREE TRIAL—GET STARTED TODAY!
www.informit.com/safaritrial

Find trusted answers, fast

Only Safari lets you search across thousands of best-selling books from the top technology publishers, including Addison-Wesley Professional, Cisco Press, O'Reilly, Prentice Hall, Que, and Sams.

Master the latest tools and techniques

In addition to gaining access to an incredible inventory of technical books, Safari's extensive collection of video tutorials lets you learn from the leading video training experts.

WAIT, THERE'S MORE!

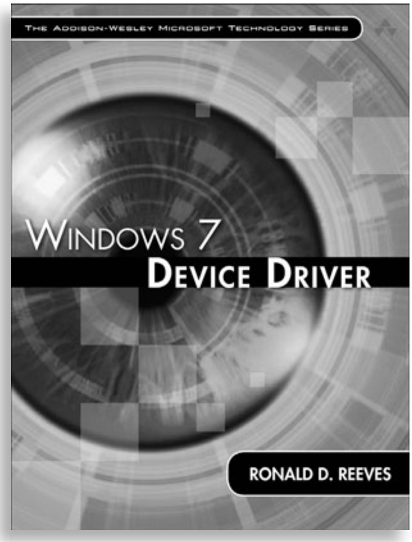
Keep your competitive edge

With Rough Cuts, get access to the developing manuscript and be among the first to learn the newest technologies.

Stay current with emerging technologies

Short Cuts and Quick Reference Sheets are short, concise, focused content created to get you up-to-speed quickly on new and cutting-edge technologies.

Addison Wesley · Adobe Press · Alpha · Cisco Press · FT Press · IBM Press · lynda.com · Microsoft Press · New Riders · O'Reilly · Peachpit Press · Prentice Hall · Que · Redbooks · Sams · sas Publishing · Sun microsystems · Wharton School Publishing · Wiley



FREE Online Edition

Your purchase of ***Windows 7 Device Driver*** includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Addison-Wesley Professional book is available online through Safari Books Online, along with more than 5,000 other technical books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly, Prentice Hall, Que, and Sams.

SAFARI BOOKS ONLINE allows you to search for a specific answer, cut and paste code, download chapters, and stay current with emerging technologies.

Activate your FREE Online Edition at www.informit.com/safarifree

STEP 1: Enter the coupon code:

STEP 2: New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com

Safari Books Online

AdobePress ALPHA Cisco Press FT Press IBM Press lynda.com Microsoft Press New Riders O'REILLY Peachpit Press PRENTICE HALL QUE Redbooks SAMS sas Publishing Sun microsystems Wharton School Publishing WILEY