The Addison-Wesley Microsoft Technology Series

Windows 7 Device Driver

Ronald D. Reeves

WINDOWS 7 DEVICE DRIVER

WINDOWS 7 DEVICE DRIVER

Ronald D. Reeves, Ph.D.

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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

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

I would like to dedicate this book to my best friend, and partner in life, my wife, Paulette. Her untiring support and love over the years have been a great source of inspiration.

This page intentionally left blank

CONTENTS

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

About the Author

Ronald D. Reeves, Ph.D., is founder and president of Software Genesis, LLC, a software development and consulting company based in Brighton, Michigan. Dr. Reeves has some forty years of experience in designing and developing computer hardware and software applications. He holds degrees in engineering and computer science and is a nationally recognized author, consultant, and teacher.

If you have questions, comments, or suggestions for improving this book, we would like to hear from you. You can contact the author by U.S. Mail or by email at the following addresses:

Dr. Ronald D. Reeves
PO Box 2425
Brighton, MI 48116
Email: software.genesis@att.net

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.

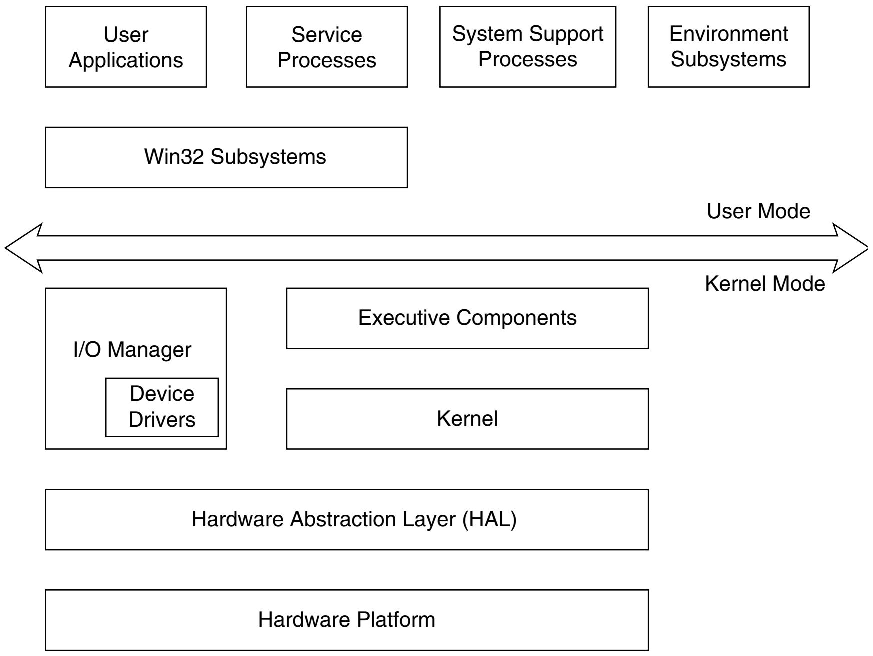

Figure I.1 System Overview Windows 7

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

PART I

DEVICE DRIVER ARCHITECTURE OVERVIEW

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.

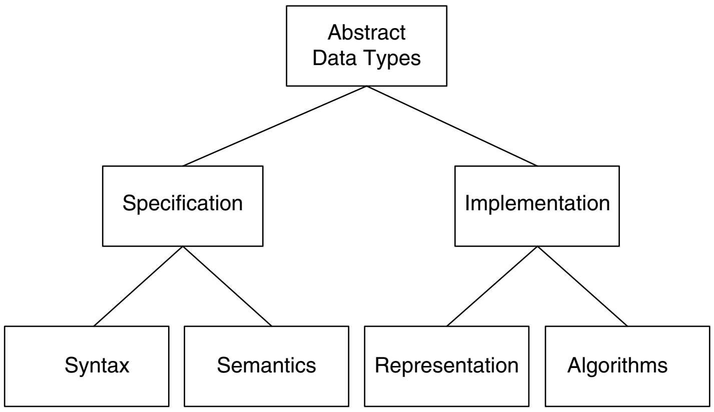

Figure 1.1 Abstract Data Type

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.

CHAPTER 2

WINDOWS DRIVER FOUNDATION (WDF) ARCHITECTURE

The next generation driver model for the Windows family of operating systems is the Windows Driver Foundation (WDF). This new model can reduce driver development time, contribute to greater system stability, and improve driver serviceability. In this chapter, we cover the overall WDF Driver Model and its various functionality. In the subsequent chapters on User Mode Drivers and Kernel Mode Drivers, we will drill down into the programming details of developing one or the other type driver. This chapter then should give a good overall feel for the general WDF driver model architecture. Note: In general, when we have a programming construct or variable, we present that information in a bold format. This of course covers the various WDF APIs available to us for developing the driver.

2.1 WDF Component Functions

WDF includes a suite of components that support the development, deployment, and maintenance of both Kernel Mode and User Mode Drivers. WDF components work with existing driver development tools to address the entire driver cycle of the following:

- Plan & Design: Driver Model—The WDF driver model supports the creation of object-oriented, event-driven drivers. By using WDF, driver writers can focus on their device hardware, rather than on the operating system. WDF drivers can be written for either kernel mode or user mode.
- Develop: Frameworks and Windows Driver Kit (WDK)—WDF defines a single driver model and includes frameworks for both Kernel Mode and User Mode Driver development. The frameworks

provide the basic infrastructure to support the WDF model. They implement common features, provide intelligent defaults, and manage most interactions with the operating system.

The Kernel Mode Driver Framework (KMDF) implements basic Kernel Mode Driver support features that are required by Windows and are common to all Kernel Mode Drivers.

The User Mode Driver Framework (UMDF) provides functional support similar to that in the KMDF, but enables drivers for some types of devices to run in user mode instead of in kernel mode.

- Test: Tracing and Static Analysis Tools—Both the KMDF and the UMDF have built-in verification code and support integrated tracing through Event Tracing for Windows (ETW). The generated traces can help in debugging drivers during development and in diagnosing problems in released drivers. WDF drivers also work with the existing driver verifier. In addition, compile-time driver verification tools, such as PREfast and Static Driver Verifier (SDV), are also part of the WDF effort.
- Qualify: Driver Signing—WDF drivers are signed in the same way as Windows Driver Model (WDM) drivers.
- Deploy: Driver Installation Tools—WDF drivers are installed by using INF files and work with existing driver installation tools, including the Driver Install Frameworks (DIFx) tools.
- Maintain: Versioning—WDF supports versioning so that a single driver binary can run on any version of the operating system and use the same version of the framework on which it was built and tested.

2.2 Design Goals for WDF

Writing a Windows driver is not easy. The current Kernel Mode Driver development model Windows Driver Model (WDM) is complex and has serious limitations.

WDM requires that drivers be designed to manage interactions with the operating system, not just the device hardware. A simple WDM driver has thousands of lines of code, much of which implements common features that every driver must support. WDM drivers must use device-driver interfaces (DDIs) that are exported directly from the operating system kernel. These interfaces were designed for performance, not for ease of use. In many cases, the DDIs expose essential operating system data structures directly to the driver, thus increasing the chance that a driver error might crash or corrupt the system.

For some device types, port/miniport models implement much of the WDM code. However, Windows supports more than 10 such models and each is different. So the knowledge gained from writing a miniport driver for one type of device does not necessarily apply to writing a miniport driver for a different type of device.

Unlike Kernel Mode Drivers, User Mode Drivers have no common infrastructure that is comparable to WDM.

The following are the primary design principles underlying the WDF model:

- Separate the driver model from the core operating system components.
- Provide a user mode option for some device types.
- Implement common and default driver features so that driver developers can focus on their hardware.
- Make drivers event driven and define the events at a detailed level so that driver tasks are straightforward.
- Support Plug and Play and power management implementation for all drivers.
- Support a consistent installation process for both User Mode and Kernel Mode Drivers.
- Provide integrated tools, including built-in tracing and verification support, to help find and diagnose problems both during debugging and after release.
- Enable a single driver binary to work with several versions of the framework and the operating system.

2.3 Device and Driver Support in WDF

Table 2.1 lists the WDF support for various device classes and driver models in Windows 7. From this table, we can get a feel for the wide range of device types that Windows 7 supports. As we have mentioned earlier, this book is primarily about creating custom device drivers. That is, ones not normally supplied by Microsoft. Notice also the distribution of device types across the two driver modes—that is, Kernel Mode Driver Framework (KMDF) and User Mode Driver Framework (UMDF).

Table 2.1 WDF Device Support for Windows 7

Device Class/Driver Model	KMDF	UMDF	SDV	PREfast
Antivirus filters	No	No	Yes	Yes
CD-ROM device	Yes	No	Yes	Yes
Cell phones	No	Yes	No	Yes
Digital cameras	No	Yes	No	Yes
Display adapters	No	No	No	Yes
DSL/Cable modems	Yes	No	No	Yes
Ethernet devices	No	No	No	Yes
Keyboards and mouse devices	Yes	No	Yes	Yes
Modems	Yes	No	Yes	Yes
Other device (not listed here) that connect to a Protocol bus such as USB or IEEE 1394	No	Yes	No	Yes
PDAs	No	Yes	No	Yes
Portable media players	No	Yes	No	Yes
Printers	No	No	No	Yes
Scanners	No	No	No	Yes
SCSI/StorePort	No	No	No	Yes
Video capture devices	No	No	No	Yes

2.4 WDF Driver Model

The WDF driver model defines an object-oriented, event-driven environment in which driver code manages device-specific features and a Microsoft-supplied framework calls the driver to respond to events that affect operation of its device. The driver model includes the following:

- An object model that is implemented by both KMDF and UMDF.
- A Plug and Play and power management implementation that both frameworks use.
- An I/O model in which the frameworks handle interactions with the operating system and manage the flow of I/O, Plug and Play, and power management requests.

- A versioning strategy that applies to both Kernel Mode and User Mode Drivers.
- Consistent installation techniques for both Kernel Mode and User Mode Drivers.

This design has several important advantages:

- The frameworks implement common driver features and default behavior, thus making vendor-written drivers smaller and faster to develop and debug.
- Microsoft can change the operating system's internal data structure without introducing driver incompatibilities.
- Driver developers and hardware vendors are better isolated from incremental changes in each new version or update of the operating system.
- Each framework can track the state of the driver, operating system, and device, thus eliminating much of the complex logic often required in a driver, particularly in respect to Plug and Play and power management.

The WDF model provides a consistent but extensible driver development interface. Both frameworks conform to conventions for naming, parameter types and usage, object hierarchy, and default. Features that are required by or common to all device types are part of each overall framework, so driver writers can apply knowledge gained from writing a driver for one device type to writing a driver for another device type.

2.5 WDF Object Model

In Chapter 1, Objects, we covered what are objects and classes. This is the point in our discussion of the WDF object model: that we start to talk about the use of objects by the WDF. Objects are a significant fundamental element of our device driver program development. Of course, many other aspects of Windows 7 use objects as well. After looking over this section, you might want to go back and revisit Chapter 1 again.

In the WDF object model:

- Objects work as building blocks for the driver. A driver modifies these objects through well-defined interfaces. The objects themselves have well-defined life cycles.
- A set of events can affect each type of object. The framework defines default behavior for each event. To support device-specific behavior, the driver includes callback routines that override the defaults.

The model defines a set of objects that represents common driver constructs, such as devices, queues, I/O requests, and the driver itself. The objects have properties, methods, and events:

- **Properties** describe characteristics of the object. Each property is associated with methods that get and (if relevant) set the value of the property.
- **Methods** perform actions on the objects.
- **Events** are conditions for which a driver might need to take action. WDF identifies possible events for each object and defines default actions for most of them. The driver includes code to handle only the events for which the default actions are inappropriate or inadequate for its device. When the event occurs, WDF invokes the related callback.

The WDF driver creates instances of the objects that it requires to service its device and customizes those instances to suit its requirements. For each instance, the driver provides callbacks for the events that require actions other than the WDF defaults. The callbacks call methods on the object to perform any additional actions.

Objects are organized hierarchically. The WDF driver object is the root object; all other objects are subordinate to it. For most types, a driver can specify the parent when it creates the object. If the driver does not specify a parent at object creation, the framework sets the parent to the WDF driver object by default. Some object types, however, have predefined parents that cannot be changed at creation. For example, I/O queue objects are children of the device object. Each child object is deleted when its parent object is deleted.

Although the object model applies to both the KMDF and UMDF, WDF objects themselves are implemented differently in the two frameworks.

2.5.1 Kernel Mode Objects

KMDF objects are structures that are opaque to the driver. Drivers never directly access instances of KMDF objects. Instead, they reference object instances by handles. To read, write, or perform an action on an object, a driver calls a method on the object and passes the handle.

The KMDF defines more than 20 types of objects. Table 2.2 lists some of the most commonly used.

KMDF objects are unique to the framework. They are not managed by the Windows object manager and therefore cannot be manipulated by using the system's **ObXxx** functions. Only the framework and WDF drivers can create and manipulate them.

Similarly, KMDF events are not related to the kernel dispatcher events that Windows uses as synchronization mechanisms. A driver cannot create, manipulate, or wait on a WDF event. Instead, the driver registers a callback for the event and WDF calls the driver when the event occurs.

2.5.2 User Mode Objects

UMDF objects are based on the component object model (COM). The UMDF uses a small subset of COM for query-interface and reference counting features. In User Mode Drivers, both the driver and the framework

Table 2.2 Commonly Used KMDF Object Types

Object Type Name	Usage
WDFDRIVER	Represents the driver object
WDFDEVICE	Represents a device object
WDFQUEUE	Represents a queue of I/O request
WDFINTERRUPT	Represents an interrupt resource
WDFREQUEST	Describes an I/O request
WDFMEMORY	Describes a buffer for an I/O request
WDFDMANENABLE	Describes the characteristic of all DMA transfers for a device
WDFDMATRANSACTION	Manages operations for an individual DMA request
WDFIOTARGET	Represents the driver that is the target of an I/O request

Table 2.3 Interfaces for UMDF Object Types

Object Interface Name	Usage
IWDFObject	Defines the base WDF object type
IWDFDriver	Represents the driver object
IWDFDevice	Represents a device object
IWDFFile	Represents a file object
IWDFIoQueue	Represents a queue of I/O requests
IWDFIoRequest	Describes an I/O request
IWDFIoTarget	Represents the driver that is the target of an I/O request
IWDFMemory	Provides access to an area of memory

implement and expose COM-style interfaces. Handles are not required because the interfaces are abstract base classes and thus identify the object.

The UMDF defines fewer objects than the KMDF because User Mode Drivers cannot directly access hardware and therefore do not perform direct memory access (DMA) or handle interrupts. Table 2.3 lists the interfaces that expose the UMDF object types.

2.6 Plug and Play and Power Management Support

Simplifying driver support for Plug and Play and power management and making it available in both kernel mode and user mode were primary design goals for WDF. Seamless handling of Plug and Play and power events is critically important to system reliability and a good user experience, but is exceedingly complex to implement correctly.

Much of this complexity occurs because drivers must determine the correct way to handle each Plug and Play or power management request. Proper handling depends on the driver's position, the device stack, the current state of its device, the current state of the operating system, and sometimes the nature of an impending state change for the device or system. Such support typically requires thousands of lines of code to handle tricky, state-dependent situations. Most drivers require code to handle requests that they don't even support.

WDF concentrates the state-tracking and decision-making logic in the frameworks, instead of requiring it in each driver. WDF support for Plug and Play and power management is based on the following principles:

- The driver should not be required to interpret or respond to every uninteresting request. Instead, the driver should be able to "opt in" and handle only the requests that are relevant to its device.
- The frameworks should provide default behavior for a rich set of Plug and Play and power features, including device stop, device removal, device ejection, fast resume, low run-time power usage, and device wake-up by external events.
- WDF actions at each point must be well-defined and predictable; in effect, a "contract" applies to each driver callback.
- Plug and Play and power management should be thoroughly integrated with other parts of the frameworks, such as queue management.
- The frameworks must support both simple and complex hardware and driver designs.
- A driver should be able to override any framework-supplied defaults.

2.6.1 Plug and Play/Power Management State Machine

Internally, WDF implements Plug and Play and power management as a state machine. Both the KMDF and UMDF use the same state machine. A driver includes callbacks so that it can perform device-specific actions at individual states in the machine. For example, a driver can provide a callback that is called immediately after its device enters the working state.

At each state transition, a predetermined set of events is valid for each type of object, and the framework invokes the driver's callbacks for these events in a defined order. Thus, a driver can assume that both the system and its device are in a particular state whenever it is asked to perform a Plug and Play or power management action.

The complicated logic that tracks system and device state is incorporated into the framework, not into the driver. This approach vastly reduces the amount of required decision-making in the driver—especially during power transitions—and eliminates much redundant code. Instead, the framework defines a state-related event and the driver optionally supplies a corresponding callback. As a result, a WDF driver includes code to handle only those events for which it requires device-specific support. All other events can be handled by WDF defaults.

Furthermore, Plug and Play and power management support are integrated throughout the framework so that other aspects of the driver operate properly when state transitions occur. For example, a driver can configure its I/O queues so that the framework stops dispatching requests while the device is in a low-power state.

2.7 Integrated I/O Queuing and Cancellation

WDF integrates Plug and Play and power management support with the queuing of I/O requests and, in turn, integrates queuing with request cancellation.

Both the KMDF and UMDF provide configurable I/O queues. The driver creates the queues and configures them for specific I/O request, power management characteristics, and dispatching requirements. The framework queues and dispatches requests according to the driver's specifications: sequentially (one at a time), in parallel (as soon as they arrive), or manually (at the driver's explicit request). When Plug and Play or power management events affect queuing, WDF can start, stop, or resume queuing as appropriate, depending on how the driver configured the queue.

Because Windows I/O is inherently asynchronous, handling the cancellation of an I/O request is often complex. The driver must cope with several potential race conditions and one or more locks, and the required code is typically scattered among several driver routines.

WDF relieves drivers of much of this burden by managing the locks for the I/O queues and by canceling queued requests without driver intervention. (A driver can, however, register for notification when a request is canceled.) By default, requests that are in a queue can be canceled. Requests that have been removed from a queue and dispatched to a driver cannot be canceled unless the driver specifically marks them so. WDF drivers that use these defaults typically require little if any cancellation code.

2.7.1 Concurrency

Managing concurrent operations is another challenge in writing a Windows driver. Because Windows is a pre-emptive, multitasking operating system, multiple threads can concurrently try to access shared data structures or resources, and multiple driver routines can run concurrently. To ensure data integrity, drivers must synchronize access to shared data structures.

WDF simplifies synchronization by implementing several internal synchronization mechanisms and by holding any required locks. In addition, WDF synchronization scope is a configurable object-based mechanism for specifying the degree of concurrency. (Synchronization scope is called the locking constraint in the UMDF.) An object's synchronization scope determines whether WDF invokes multiple event callbacks on the object concurrently. Drivers that use the KMDF can specify synchronization scope for driver, device, and file objects. In the UMDF, synchronization scope applies only to device objects.

WDF defines the following synchronization scopes:

- Device scope—WDF does not call certain I/O event callbacks concurrently for an individual device object or any file objects or queue objects that are its children.
- Queue scope—These I/O callbacks are not called concurrently on a per-queue basis. If a Kernel Mode Driver specifies queue scope for a device object, these callbacks can run concurrently for multiple queues. However, multiple callbacks for an individual queue object will not be called concurrently. The initial UMDF release does not support queue scope.
- No scope—WDF does not acquire any locks and can call any event callback concurrently with any other event callback.

By default, the KMDF uses no scope. A Kernel Mode Driver must "opt in" to synchronization for its objects by setting device scope or queue scope when it creates an object. The UMDF uses device scope by default.

For Kernel Mode Drivers, the KMDF also enables driver writers to constrain the interrupt request level (IRQL) at which the callbacks can be invoked.

2.7.2 I/O Model

In Windows, the I/O request packet (IRP) does more than just present traditional I/O requests (read, write, create, and so forth) to drivers. It works as a general packet-based communication mechanism between the operating system and drivers, and between drivers themselves. The Windows I/O manager sends IRPs to notify drivers of Plug and Play requests, power management requests, changes in device status, and queries about device and driver resources (among other purposes) in addition to passing I/O requests. Therefore, the WDF I/O model encompasses more than just data transfers to and from a device.

For WDF drivers, the framework manages the mechanics of dispatching, queuing, completing, and canceling IRPs on behalf of its drivers. The framework calls the driver's event callback routines to notify it of significant events such as requests that the driver must handle.

After receiving a request, the framework records information about the request, creates a WDF object to represent the request (if necessary), and calls one or more of the driver's event callbacks to handle the request as appropriate. WDF queue objects help drivers to manage the arrival of I/O requests. A driver can create one or more such queues and configure each to receive specific types of requests. Depending on the dispatch mechanism that the driver has designed for each queue, the framework either delivers the request to the driver immediately or queues it for later delivery.

The framework keeps track of every I/O request, whereas the driver "owns" the request—that is, until the request has been canceled, completed, or passed to another target. Because the framework is aware of all the active requests, it can call the appropriate driver callbacks in case of IRP cancellation, power state changes, hardware removal, and so forth.

2.7.3 I/O Request Flow

Both the KMDF and UMDF use the same I/O model, although it is implemented by different components. Within this model, I/O request flow is as shown in Figure 2.1.

As Figure 2.1 shows, WDF dispatcher code directs I/O request packets within the framework. WDF dispatches I/O requests according to their major I/O function code. The major function code is a field within the IRP that identifies the type of request. Based on the major I/O function code, the dispatcher determines which package within the framework should initially handle the request.

The following sections describe how WDF processes requests.

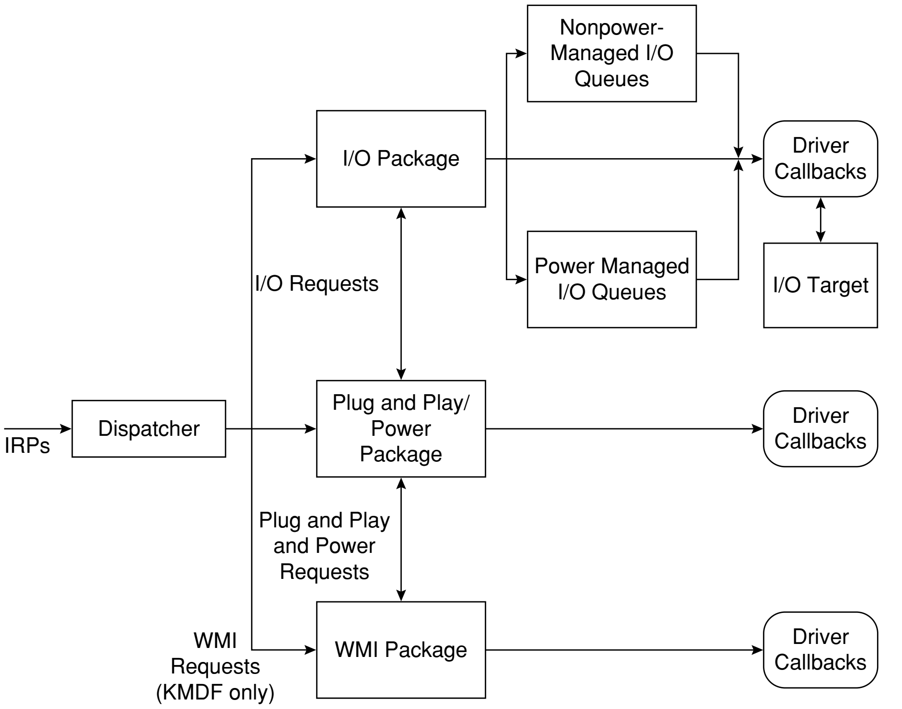

Figure 2.1 Block Diagram I/O Request Flow

2.7.4 Device I/O Requests

When an IRP that requests device I/O arrives, the dispatcher passes it to the I/O package. If the driver has not configured a queue or exposed a callback for the requested type, the framework takes a default action that depends on the type of driver. For a User Mode Driver, or for a Kernel Mode Function Driver or Bus Driver, the framework fails the request. For a Kernel Mode Filter Driver, the framework forwards the request to the next lower driver in the stack.

If the driver has configured a queue or exposed a callback for the request type, the framework creates a WDF request object, which contains the information in the original IRP structure along with additional information about the driver state. The framework then places the request object in the corresponding queue.

If the queue is configured for automatic power management, the framework then determines whether the device is in the correct power state. If not, the Plug and Play and power package puts the device in the working state. If the driver has registered callbacks for power events, the framework calls the Plug and Play; otherwise, it takes whatever default steps are required.

After the device has entered the working state, the framework dispatches the I/O request according to the driver's specifications by invoking the callbacks registered for the I/O request. A driver can also request manual dispatching, which means that it must call the framework to get a request. The framework passes the WDF request object when it invokes the callbacks. The driver's callbacks might set or get properties for the request, call methods on the request object or other WDF objects, perform device I/O, and take other actions as necessary to handle the request.

When the driver has finished processing the request, the driver can complete it or pass it on to an I/O target. An I/O target is an external destination for the I/O request. The next lower driver in the device stack is considered the local I/O target; any other driver is considered a remote I/O target.

If a driver does not complete an I/O request, it typically sends the request to its local I/O target. Occasionally, however, a driver might require information from a different driver before it can complete a request. To obtain this information, the driver creates an object to represent the remote I/O target, creates a WDF request object, and then calls methods on the I/O target to send the request.

2.7.5 Plug and Play and Power Management Requests

When a Plug and Play or power request arrives, the framework determines whether any Plug and Play or power management state changes are required to satisfy the request. If so, the framework takes the necessary actions to change the state and either calls the driver's registered event callbacks or performs default actions if the driver has not registered any callbacks for those events.

After the relevant callbacks have returned, the framework completes or forwards the request, as appropriate, on the driver's behalf.

2.8 WMI Requests (Kernel Mode Drivers Only)

A Windows Management Instrumentation (WMI) request triggers callbacks that the driver registered for any current WMI events. In its WMI callbacks, the driver might call WMI methods on the device object to create and manipulate WMI instances of to change its status as a WMI provider. After the WMI callbacks have returned, the framework completes or forwards the request, as appropriate, on the driver's behalf. Only the KMDF supports WMI.

To understand how an I/O request flows through a WDF driver, consider the following scenario:

- A user mode process requests a read from a device.
- At the time of the request, the device is in a low-power state.
- The driver has configured a power-managed queue to accept read requests.

The request is processed by the WDF function driver as follows:

1. The IRP dispatcher inspects the IRP and directs it to the I/O package. The I/O package creates a WDF request object to represent the IRP, adds the WDF request object to the queue, and checks the current device power state. Because the device is in a low-power state, the I/O package calls the Plug and Play/power management package to put the device in the fully powered working state so that it can perform the read operation.
2. The Plug and Play/power management package returns the device to the working state by taking default actions and calling the appropriate power management callbacks implemented by the driver.
3. When the device has successfully reentered the working state, the framework dispatches the read request to the driver. If the driver has configured manual dispatching, the driver calls a method on the queue to get a request. Otherwise, the framework dispatches the request either immediately or when the driver has completed the previous request, depending on the queue's configuration.
4. If the driver can satisfy the request, it does; if it cannot, it sends the request to an I/O target.

2.9 Driver Frameworks

The WDF driver model is implemented through the KMDF, which supports Kernel Mode Driver development, and the UMDF, which supports User Mode Driver development. The frameworks provide the basic driver infrastructure and perform the following services for WDF Drivers:

- Define WDF objects that drivers can instantiate.
- Manage object lifetimes.
- Expose a basic set of DDIs that drivers call to manipulate the objects.
- Provide a common implementation of features that drivers typically require, such as Plug and Play, power management, synchronization, I/O queues, and access to the registry.
- Manage the flow of I/O requests and Plug and Play and power notifications from the operating system to the driver.

Instead of calling the operating system directly, drivers interact with the appropriate framework for most services. The frameworks manage most of the interactions with the operating system on behalf of the driver. In effect, the frameworks shield driver developers from the details of the operating system.

The frameworks implement the WDF I/O model, object model, and Plug and Play and power management support. Each framework receives I/O requests, calls the driver to handle events according to the driver's configuration, and applies defaults otherwise. Both frameworks provide intelligent defaults for common operations so that drivers do not require large amounts of potentially buggy "boilerplate" code.

The frameworks support common features required for all device classes. Device-class-specific extensions can also be added. For example, the initial release of the KMDF supports extensions specifically for USB devices. As new features are added to the operating system, and as new device classes are supported, features that are common to all device classes will be added to the base set of DDIs in the frameworks. Extensions will provide features that are required by one or more specific device classes, but not by every device class. The extensions are intended to replace the miniport models common with WDM.

2.9.1 Kernel Mode Framework

For Kernel Mode Drivers, the KMDF does not replace WDM; instead, it provides a skeletal WDM implementation. In effect, the driver developer configures the skeletal driver to work with a particular device by creating objects and providing event-based callback routines.

The KMDF is a reentrant library that can be shared by multiple drivers. Drivers are dynamically bound with the library at load time, and multiple versions of the library can be used by multiple drivers simultaneously.

The KMDF currently supports creation of the following types of Kernel Mode Drivers:

- Function drivers for Plug and Play devices.
- Filter drivers for Plug and Play devices.
- Bus drivers for Plug and Play device stacks.
- Control device drivers for legacy (NT 4.0-style) devices that are not part of a Plug and Play stack.

Currently, the KMDF does not support bus filter drivers.

WDF provides certain methods and callbacks specifically for bus drivers, others specifically for function and filter drivers, and still others for control device drivers.

The KMDF identifies a function driver, control device driver, or a bus driver based on the methods that the driver calls and the callbacks that the driver supports. For example, the bus driver for a device typically supports callbacks to enumerate the children of the device and to supply a list of the hardware resources that the device requires. A function driver for a device typically supports callbacks to manage power to its device.

A filter driver explicitly identifies itself as such before creating a device object. The KMDF uses this information when passing I/O requests to the driver. A filter driver registers for only the I/O requests it chooses to filter; the KMDF passes all other requests to the next lower driver. (For a function or bus driver, WDF fails other requests.) By contrast, a WDM filter driver must accept all I/O requests that could be targeted to its device, pass those it does not filter to a lower driver, and act on the remaining subset. A WDM filter driver requires logic to inspect and forward many types of requests; a WDF filter driver has no such code because it receives only the requests it is interested in.

When an application sends an I/O request to a Kernel Mode WDF Driver, the request travels through the components shown in Figure 2.2.

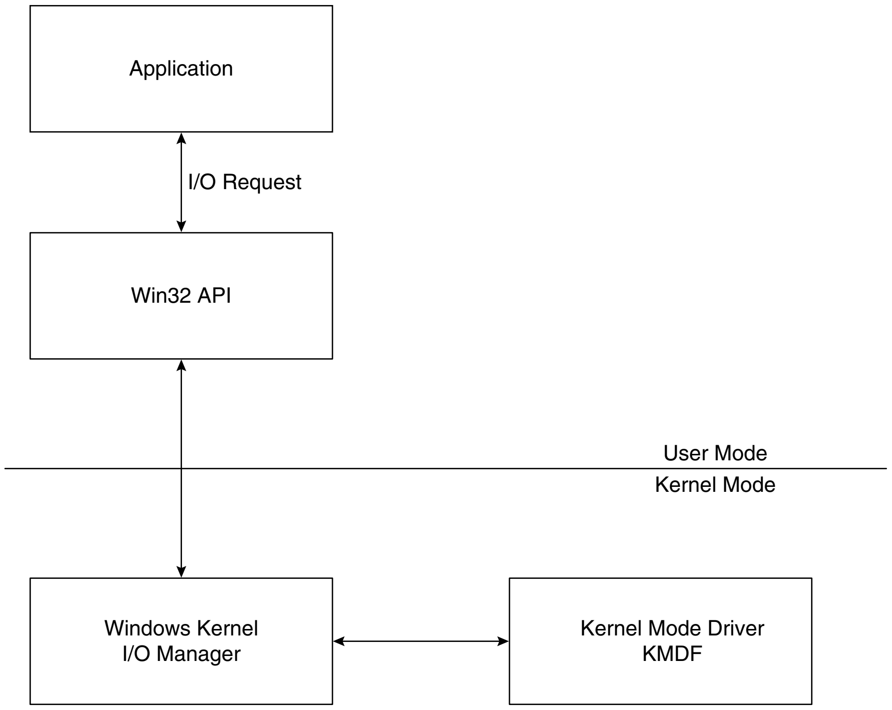

Figure 2.2 I/O Flow to Kernel Mode WDF Driver

As the figure shows, the following components are involved in handling an I/O request to a Kernel Mode WDF Driver:

- **Application**—The application is a user mode process that issues I/O requests through the Win32 API.
- **Win32 API**—In response to the application's I/O request, the Win32 API calls I/O routines in the Windows kernel.
- **Windows kernel**—The I/O manager in the Windows kernel creates an IRP to represent the request and presents it to the target driver by calling the driver at a designated entry point. For Kernel Mode WDF Drivers, the KMDF registers the entry points, in effect intercepting the request on behalf of the driver.
- **KMDF**—The KMDF processes the request as previously described in the section "I/O Request Flow," creating a WDF request object and calling the driver's event callback routines as required.

2.9.2 User Mode Framework

The UMDF implements a subset of the KMDF functionality, including support for Plug and Play, power management, and asynchronous I/O. Drivers that run in user mode have access only to the user address space and therefore pose low risk to system stability. User Mode Drivers cannot handle interrupts, perform DMA, or use kernel mode resources such as nonpaged pool.

Using the UMDF, developers can create drivers for any protocol or serial-bus based device. Although these drivers run in user mode, they use the standard Plug and Play installation mechanism and the same I/O model as Kernel Mode WDF Drivers. Figure 2.3 shows the components involved in transmitting an I/O request from an application to a User Mode WDF Driver.

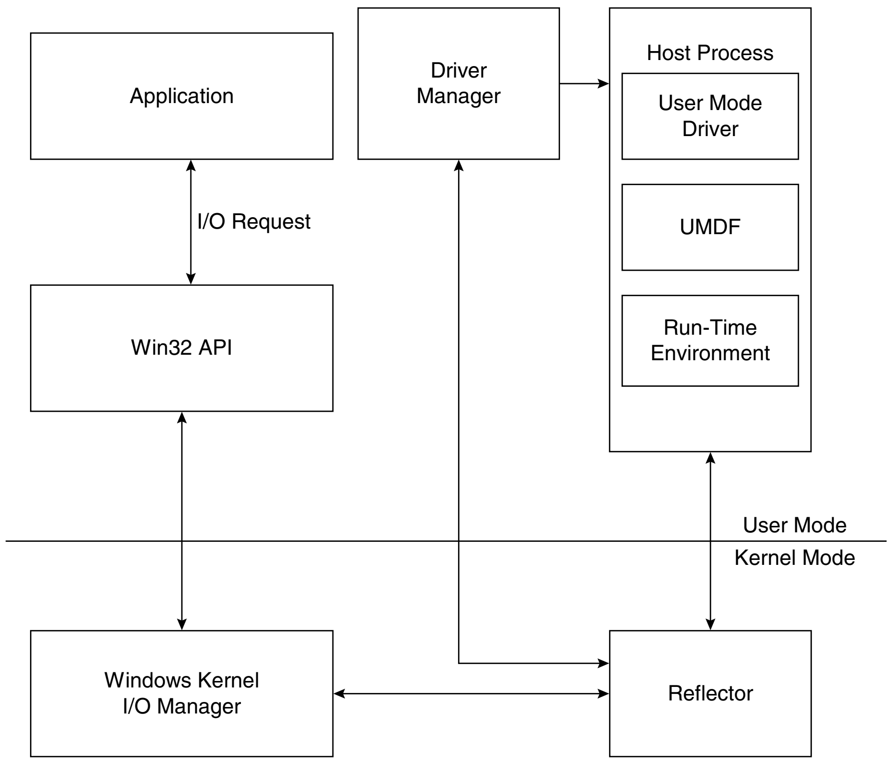

Figure 2.3 I/O Flow to User Mode WDF Driver

Figure 2.3 includes the following components, described according to the typical flow of an I/O request:

- **Application**—The application is a user mode process that issues I/O requests through the Win32 API.
- **Win32 API**—In response to the application's I/O request, the Win32 API calls I/O routines in the Windows kernel.

2.10 Windows Kernel

The I/O manager in the Windows kernel creates IRPs to represent the requests and presents them to the target driver by calling the driver at a designated entry point. If the target of the request is a User Mode WDF Driver, however, the I/O manager cannot call the driver or the UMDF directly because these components run a user mode process, and kernel mode components cannot be called back to user mode. Therefore, the I/O manager does not present the request directly to the User Mode Driver. Instead, the I/O manager presents the request to a kernel mode component called the reflector.

2.10.1 Reflector

The reflector is a Kernel Mode WDM Filter Driver that represents the User Mode Driver in the Kernel Mode Driver stack. The reflector passes the I/O request to the User Mode Driver host process.

The reflector manages communication between the kernel mode components and the User Mode Driver host process. It monitors the driver host process to ensure that it responds properly to messages and completes critical operations in a timely manner, thus helping to prevent driver and application hangs. The reflector also sends messages to the driver manager as required.

The reflector is supplied by Microsoft and is added as the top driver in the Kernel Mode Driver stack during installation of the User Mode Driver.

2.10.2 Driver Host Process

The driver host process is the user mode process in which the User Mode Driver runs. It includes the following components:

- The User Mode WDF Driver is an in-process COM component that controls the hardware from user mode.

- The UMDF exposes the User Mode DDI. The UMDF is a dynamic-link library (DLL) of COM-style objects that support the presentation, flow, and management of I/O, Plug and Play, and power management requests to the driver.
- The run-time environment dispatches I/O requests, loads the driver, constructs and destroys the user mode device stack, manages a user mode thread pool, and handles messages from the reflector and the driver manager.

The driver host process is separate from the application process and the driver manager. It runs in the security credentials of a LocalService account, although it is not a Windows service. The driver host process contains the user mode device stack for the device. The device stack is visible to all applications across the system. Each instance of a device has its own device stack. Currently, each instance has a separate driver host process, too. The driver host process is a child process of the driver manager.

2.10.3 Driver Manager

The driver manager creates and shuts down the driver host process and maintains status information about it. It also responds to messages from the reflector. The driver manager runs as a Windows service is started during installation of the first device that is managed by a User Mode WDF Driver. The driver manager must be running all the time that any device controlled by a User Mode WDF Driver is installed on the system.

2.11 Tools for Development and Testing

The WDF has some outstanding tools to aid in the testing of drivers. We will cover those in the following discussion. Thoroughly testing a driver is nearly as complex as writing one for two main reasons:

- Observing the point of error can be difficult. In many cases, a driver error is not apparent until long after it has actually occurred. If a Kernel Mode Driver uses a DDI incorrectly, the system might not crash until another driver attempts to perform an action based on the first driver's error.
- Subtle, condition-dependent errors and related code paths are difficult to exercise. Drivers that work correctly under normal circumstances

can have subtle errors that occur only under exceptional situations, such as when another driver, lower in the stack, fails an I/O request.

Too often, testing becomes a hit-or-miss, trial-and-error affair. To help remedy the situation, WDF has several testing and tracing features that make it easier for driver writers to find problems early in the development cycle. We will cover these features in more detail in the subsequent discussion. These features include the following:

- Built-in verification with the frameworks verifier
- Built-in trace logging
- Debugger extensions

In addition, WDF includes PREfast and Static Driver Verifier (SDV). PREfast and SDV are both compile-time code verification tools that are provided with the Windows Driver Kit (WDK). PREfast analyzes code on a function-by-function basis, looking for a wide variety of common logic and usage errors. SDV applies knowledge about system internals to Kernel Mode Driver verification.

2.11.1 PREfast for Drivers

PREfast for Drivers (PFD), an extension of PREfast, is a compile-time static verification tool that detects errors missed by the compiler and by conventional run-time testing. It detects common coding errors in C and C++ programs, and is designed to detect errors in Kernel Mode Driver code. You can run PFD very early in the development cycle—as soon as the code compiles correctly. PFD is integrated into the Windows 7 build environments in the Windows Driver Kit (WDK) as well as into Windows Automated Code Review (known as OACR). PFD supports a large vocabulary of annotations beyond those supported for generic PREfast, including annotations for IRQLs, resource-object leaks, memory leaks, and stricter type checking.

PREfast for Drivers examines each function in the code independently, looking for common errors and unwise coding practices. PFD runs quickly, even on large drivers, and generates a report that identifies the line of driver code with the suspected error.

PREfast for Drivers runs on Windows XP and later versions of Windows and is designed to analyze code written for X86-based and

X64-based platforms. It can analyze C and C++ source files for drivers in any driver model, including managed code.

You should use PREfast for Drivers in conjunction with Driver Verifier, Static Driver Verifier, and the checked build of Windows to ensure that your driver code is safe and reliable.

The following new features for PFD are in Windows 7:

- PFD now supports a broader range of expressions for analysis, such as const, member names, and side effect-free C expressions.
- PFD now has better annotation error checking.
- PFD now has improved defect detection, including "banned API" checking.
- PFD now generates warnings that help you prepare to analyze a driver with Static Driver Verifier (SDV). SDV requires drivers to have declarations that define the role of the driver-supplied callback functions. PFD will indicate when you need to add these role type declarations to the drive code.
- PFD is now integrated into the build environments and OACR in the WDK. When you build your driver using the WDK build environments, PFD runs automatically in the background and presents an easy-to-read view for any potential defects it finds.

For Windows 7, all Microsoft drivers that ship with the operating system and all WDK samples have been verified with PFD, and identified defects have been fixed. In addition, the WDK public headers are now annotated to enable PFD to better find code defects. The following functionality aids in finding code defects.

- Because PFD annotations are not in public header files, driver writers can take advantage of these checks by simply running PFD on their drivers. Adding PFD annotations to your driver code will give you deeper analysis.
- Windows headers for drivers now provide a comprehensive set of examples of how to annotate your functions.

2.11.2 Static Driver Verification (SDV)

Static Driver Verifier (SDV) is a static verification tool that runs at compile time. It explores paths in the driver code by symbolically executing the source code, making the fewest possible assumptions about the state of the operating system and the initial state of the driver. As a result, SDV can exercise code in paths that are missed in traditional testing.

SDV includes a set of rules that defines proper interaction between a driver and the operating system kernel. During verification, SDV examines every applicable branch of the driver code and the library code that it uses, and tries to prove that the driver violates the rules. If SDV fails to prove a violation, it reports that the driver complies with the rules and passes the verification.

2.11.3 Frameworks Verifier

WDF includes an internal driver verifier that provides framework-specific features that are not currently available in the driver verifier (Verifier.exe). The frameworks verifier provides extensive tracing messages that supply detailed information about activities within the framework. It tracks references to each WDF object and builds a trace that can be sent to the debugger.

In kernel mode, the frameworks verifier checks lock acquisition and hierarchies, and ensures that calls to the framework occur at the correct IRQL. It also verifies correct I/O cancellation and queue usage. It can also simulate low-memory and out-of-memory conditions and test a driver's response to these situations to determine whether the driver responds properly without crashing, hanging, or failing to unload.

In user mode, the frameworks verifier checks for correct use of parameters, validates configurations, and correct responses to events.

2.11.4 Trace Logging

Both the KMDF and UMDF support integrated internal trace logging. The following discussions cover this internal trace logging.

The KMDF includes an internal trace logger called the in-flight recorder (IFR), which is based on the Windows Software Trace Preprocessor (WPP). The IFR provides a recent history of events (currently, about the last 100 trace events) on a per-driver-instance basis. The trace logs track the progress of IRPs through the framework and the corresponding requests through a driver. Each WDF driver has its own log.

Kernel Mode Drivers can use Event Tracing for Windows (ETW) and WPP software tracing to generate a trace log that contains information about both the driver and the KMDF. Driver-level tracing provides information about events in the driver code. Internal WDF tracing provides information about events internal to WDF that might affect driver activities. A driver developer can choose whether to implement driver-level tracing, but internal WDF tracing is always available.

Driver writers can use the software tracing tools provided with the WDK to view the IFR logs during interactive debugging. These logs can also be made available as part of a mini dump for inspection after a crash. The typical saved IFR log file is small (10K to 20K bytes) and written in a binary form that humans cannot read.

The User Mode Driver components supplied by Microsoft start trace sessions that record their activities and note such events as driver hangs, timeouts, and failures. The log files from these sessions can be sent as input to Windows Error Reporting (WER). Vendor-supplied User Mode WDRF Drivers can use ETW to generate a trace log of driver events.

2.11.5 Debugger Extensions

WDF also includes several debugger extensions that can dump internal trace records. These extensions are specialized commands that run in the context of the WinDbg debugger. These extensions are packaged in two DLLs: WudfExt.dll contains the UMDF extensions, and WdfKd.dll contains the KMDF extensions. The information they provide can help locate the exact point in I/O processing at which an error occurred and can often give a clue to faulty assumptions or unexpected behavior.

The two sets of debugger extensions are provided for WDF. As mentioned, one set supports user mode debuggers, and the other supports kernel mode debuggers.

2.11.6 Serviceability and Versioning

To improve driver serviceability, WDF includes versioning and side-by-side support. Versioning allows a driver binary to run with the same major version of WDF with which it was built. Side-by-side support enables the simultaneous use of two or more major versions of WDF by two or more drivers.

Serviceability is a common problem for drivers. When Microsoft releases a new version of Windows, driver vendors must test their drivers to ensure that they operate properly on the new release. Any driver that uses undocumented features, or that uses documented features in a non-standard way, is likely to encounter compatibility problems from one release to the next. Even drivers that follow the rules might be affected by subtle changes between versions of Windows.

Drivers that use the frameworks, however, are less susceptible to such problems. Microsoft is responsible for testing the frameworks on each new version of the operating system and ensuring that drivers built with older versions maintain consistent behavior from one release to the next.

In addition, the versioning support in WDF helps to prevent compatibility problems. The frameworks have major and minor version numbers, which are recorded in the driver binaries. In general, a WDF driver runs against the latest available minor version of the major version against which it was compiled, so that it can benefit from bug fixes in the new version.

A WDF driver can use a newer minor version, but not an older minor version, than the one against which it was built. Multiple WDF drivers can use a single WDF library. They can also run side by side using different major versions of the framework.

PART II

USER MODE DRIVERS

This page intentionally left blank

CHAPTER 3

WINDOWS 7 USER MODE DRIVERS OVERVIEW AND OPERATION

The Windows Driver Foundation (WDF) contains a framework for the creation of User Mode Drivers. The User Mode Driver Framework (UMDF) is designed to support protocol device classes such as cameras and portable music players. It integrates the installation and management of these devices with standard operating system facilities, such as I/O and Plug and Play and power management.

UMDF is based on the same conceptual driver programming model as the Kernel Mode Driver Framework (KMDF) that is also part of WDF. However, the two frameworks implement the model with different components, device-driver interfaces (DDIs), and data structures. KMDF includes some objects that are available only in kernel mode, and UMDF includes some objects that are available only in user mode.

Like KMDF, UMDF provides intelligent defaults, so that driver developers can focus on their device hardware and avoid writing code to perform many common driver tasks. Instead, that code is built into the framework, thus making vendor-written drivers smaller, ensuring greater code reuse, and providing for global bug fixes by Microsoft.

This chapter describes the architecture and features of UMDF and outlines the requirements for drivers that use UMDF (sometimes called UMDF-based drivers or simply UMDF drivers).

3.1 Devices Supported in User Mode

UMDF supports the development of drivers for protocol-based or serial bus-based devices, such as Universal Serial Bus (USB) devices and network-connected devices. For example, drivers for the following types of devices can be written in user mode:

- Portable storage devices
- Portable media players
- USB bulk transfer devices
- Auxiliary display devices

The device can be directly connected, connected on the network, or connected via a wireless protocol such as Bluetooth. UMDF also supports software-only drivers.

The initial UMDF release includes the following sample UMDF drivers:

- Skeleton—A minimal driver that is intended for use as a template for driver development.
- Echo—A simple-software-only driver that shows the use of a serial I/O queue.
- USB/FX2_Driver and USB/Echo Driver—Function drivers for the USB-FX2 board that was designed by Open Systems Resources, Inc. (OSR). This is the board we will be using for our driver example developments for UMDF.
- USB/Filter—A filter driver for the USB-FX2 device stack.

User Mode Drivers can support 32-bit or 64-bit devices for any Windows hardware platform and can be distributed on Windows Update. UMDF is currently supported for Windows 7, Windows Vista, and Windows XP.

Drivers that require the following cannot be written as UMDF drivers; they must be written as Kernel Mode Drivers:

- Handling interrupts
- Direct access to the hardware, such as direct memory access (DMA)
- Strict timing loops
- Use of nonpaged pool or other resources that are reserved for kernel mode.

In addition, a UMDF driver cannot be a client of the Windows kernel or of a Kernel Mode Driver.

3.2 UMDF Model Overview

A UMDF driver runs in a driver host process that also hosts UMDF and a run-time environment. Each such driver operates as part of a stack of drivers that manage a device. The User Mode Drivers are loaded above the Kernel Mode Drivers at the top of the stack. Because user mode components do not have access to the system address space where the system and Kernel Mode Drivers maintain I/O requests and other shared data, the UMDF architecture includes components that communicate between kernel mode and user mode. Figure 3.1 shows the overall architecture of the UMDF driver model.

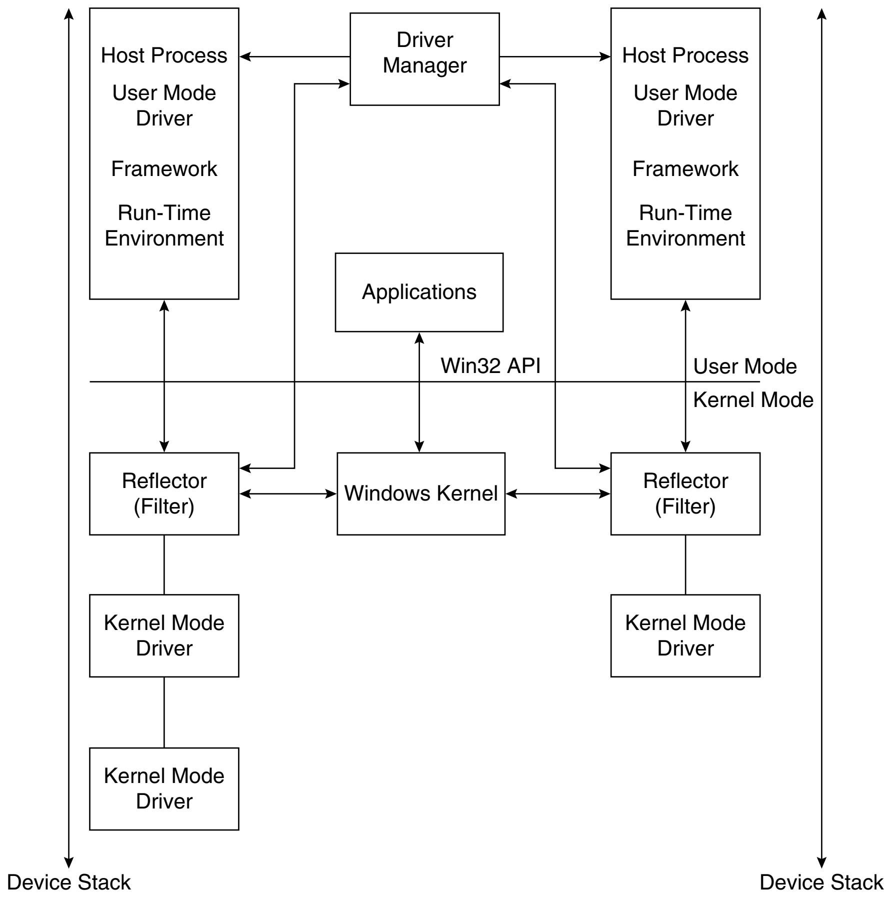

Figure 3.1 UMDF Driver Architecture

Figure 3.1 shows two device stacks that service two different devices. Each device stack includes a UMDF driver that runs in its own driver host process. The figure includes the following components, described according to the typical flow of an I/O request.

Applications. The applications are clients of the drivers. These applications are user mode processes that issue I/O requests through the Win32 File I/O API. The Win32 functions call I/O routines in the Windows kernel.

Windows kernel. The Windows kernel creates I/O request packets (IRPs) to represent the user mode I/O requests and forwards them to the top of the Kernel Mode Driver stack for the target device.

Reflector. The reflector is a Kernel Mode WDM Filter Driver that is installed at the top of the kernel mode device stack for each device that a UMDF driver manages. The reflector manages communication between the kernel mode components and the User Mode Driver host process. The reflector forwards I/O, power, and Plug and Play messages from the operating system to the driver host process, so that User Mode Drivers can respond to I/O requests and participate in Plug and Play device installation, enumeration, and management. The reflector also monitors the driver host process to ensure that it responds properly to messages and completes critical operations in a timely manner, thus helping to prevent driver and application hangs. Microsoft provides the reflector.

Driver manager. The driver manager creates and shuts down the driver host processes and maintains status information about them. It also responds to messages from the reflector. The driver manager runs as a Windows service and is started during installation of the first device that has a UMDF driver. One instance of the driver manager handles all of the driver host processes. The driver manager must be running all of the time that any device controlled by a UMDF driver is installed on the system. Microsoft provides the driver manager.

Host process. The host process is the process in which the User Mode Driver runs. It is separate from the application process and the driver manager. It runs in the security credentials of a LocalService account, although it is not a Windows service. The host process contains the user mode device stack for the device. The device stack is visible to all applications across the system. Each instance of a device has its own device stack. Currently, each instance has a separate driver host process, too.

The host process includes the following components:

- The UMDF driver is an in-process component object model (COM) component that controls the hardware from user mode.

- The framework exposes the user mode DDI, which is a dynamic-link library (DLL) of COM-style objects that support the presentation, flow, and management of I/O, power, and Plug and Play requests to the driver.
- The run-time environment dispatches I/O requests, loads the driver, constructs and destroys the user mode device stack, manages a user mode thread pool, and handles messages from the reflector and the driver manager.

The host process is a child process of the driver manager.

Kernel Mode Drivers. Additional Kernel Mode Drivers can service each device. These device drivers are supplied either by Microsoft or by the device writer.

3.2.1 UMDF Object Model

UMDF drivers are object oriented and event driven. The driver and the framework create instances of objects that are required to support the driver's device. The driver implements event callback interfaces to handle events that affect these objects.

The objects and interfaces are based on the COM programming pattern. UMDF uses only a small subset of COM, specifically the COM lifetime model; it does not depend on the entire COM infrastructure and run-time library. The UMDF run-time environment loads the driver by reading information that is stored in the registry under the WDF service key.

UMDF uses only the query-interface and reference-counting features of COM. Every UMDF interface derives from **IUnknow** and therefore supports the **QueryInterface**, **AddRef**, and **Release** methods by default. The **AddRef** and **Release** methods manage object lifetime. The **QueryInterface** method enables other components to determine which interfaces the driver supports.

3.2.2 UMDF Objects

UMDF manages a series of objects that are exposed to the User Mode Driver. UMDF creates some of these objects in response to application-triggered actions, such as an I/O request; the driver creates other objects by calling methods on UMDF interfaces.

For each type of object, UMDF defines one or more interfaces through which to manipulate instances of the object. The interfaces provide methods and properties. Methods define actions that can be taken on behalf of the object and return a status to indicate whether they succeeded or failed. Property operations set and get the attributes of the object and cannot fail. Some interfaces are implemented by UMDF, and others are implemented by the driver.

Table 3.1 lists all the UMDF object types and the interfaces that UMDF implements on each type.

Table 3.1 UMDF Object Types

Type of Object	Interfaces	Description
Base object	**IWDFObject**	Exposes a base object for use as the driver requires.
Device	**IWDFDevice**	Exposes an instance of a device object. A driver typically has one device object for each device that it controls.
Driver	**IWDFDriver**	Exposes the driver object itself. Every driver has one driver object.
File	**IWDFFile**	Exposes a framework file object that was opened by the Win32 **CreateFile** function, through which applications can access the device.
	IWDFDriverCreatedFile	Exposes a framework file object that the driver created.
I/O queue	**IWDFIoQueue**	Exposes an I/O queue, which controls the flow of I/O in the driver. A driver can have any number of I/O queues.
I/O request	**IWDFIoRequest**	Exposes a request for device I/O.
I/O target	**IWDFIoTarget**	Represents the next-lower driver in the device stack, to which the driver sends I/O requests.
Memory	**IWDFMemory**	Exposes memory that the driver uses, typically an input or output buffer that is associated with an I/O request.
USB device	**IWDFUsbTargetDevice**	Exposes a USB device object that is an I/O target. Inherits from **IWdfIoTarget**.
USB interface	**IWDFUsbInterface**	Exposes an interface on a USB device.
USB pipe	**IWDFUsbTargetPipe**	Exposes a USB pipe that is an I/O target. Inherits from **IWdfIoTarget**.

The driver calls methods on these interfaces to perform operations on its objects. For example, UMDF implements the **IWDFIoRequest** interface, and the driver calls methods in this interface to retrieve the parameters for the I/O request.

For the driver, devices, and queues, both the framework and the driver maintain objects. The driver-created objects are callback objects, on which the driver implements the callback interfaces that are required to service its device. A driver has one callback object, one device callback object for each device that it supports, and one queue callback object for each queue that it creates. The callback objects serve as the “context memory” for the driver.

3.3 Driver Callback Interfaces

The driver implements callback interfaces to provide device-specific responses to events. Each callback interface is implemented on a specific object type. For example, Plug and Play callback interface (**IPnpCallback**, **IPnpCallback-Hardware**, and **IPnpCallbackSelfManagedIo**) are implemented for device objects, and I/O queue callback interfaces (**IQueueCallbackCreate**, **IQueueCallbackRead**, and so forth) are implemented on I/O queue objects.

When a Plug and Play, power management, or I/O request arrives, UMDF calls methods in the driver’s callback interfaces to handle the associated events. For example, when UMDF receives a read request, it calls methods in the driver’s **IQueueCallbackRead** interface.

A driver implements callback interfaces only for the events that are important to its operation. When the event occurs for an object, the framework invokes the callback for that object. For example, the unexpected removal of a device is a Plug and Play event. If a device can be removed unexpectedly, its driver should implement the **IPnpCallback** interface (which includes the **OnSurpriseRemoval** method) to perform device-specific operations upon ejection. When the Plug and Play manager sends a surprise-removal notification for the device, UMDF calls the **OnSurpriseRemoval** method with a pointer to the **IWDFDevice** interface for the device that has been removed.

For most events, a driver can either provide a callback interface or allow UMDF to perform a default action in response. For a few events, however, a driver-specific callback is required. For example, adding a device is an event for which every Plug and Play driver must include a callback. The driver object's **IDriverEntry::OnDeviceAdd** callback creates the device object.

The names of the driver-implemented callback interfaces are generally in the form **IObjectAction**, where **Object** identifies the object to which the interface applies and **Action** indicates what the interface does. For example, the **IQueueCallbackRead** interface is implemented for I/O queues and contains methods called when a queue receives a read request.

Table 3.2 lists the possible callback interfaces a driver might implement.

Table 3.2 Driver Callback Objects and Interfaces

Type of Object	Callback Interfaces	Description
Base object, or any object that inherits from the base object type	**IObjectCleanup**	Provides process that is required before an object is deleted, typically releasing any references held on the object.
Driver	**IDriverEntry**	Provides main entry point and methods to initialize the driver and add its devices.
Device	**IPnpCallback**	Handles device stop, removal, and power state changes.
	IPnpCallbackHardware	Provides hardware-related operations before device power-up and after device power-down.
	IPnpCallbackSelfManagedIo	Provides driver, rather than framework, control over I/O operations at specific Plug and Play and power management states.
	IFileCallbackCleanup	Handles clean-up requests for file objects on a specific device.
	IFileCallbackClose	Handles close requests for file objects on a specific device.
I/O queue	**IQueueCallbackCreate**	Handles file create requests.
	IQueueCallbackDefault-IoHandler	Handles create, device I/O control, read, and write requests for which no other interface has been implemented.

Type of Object	Callback Interfaces	Description
I/O request	**IQueueCallbackDevice-IoControl**	Handles device I/O control requests.
	IQueueCallbackIoResume	Resumes processing an I/O request after its queue has been stopped.
	IQueueCallbackIoStop	Stops processing an I/O request because its queue is stopping.
	IQueueCallbackRead	Handles read requests.
	IQueueCallbackWrite	Handles write requests.
	IImpersonateCallback	Provides impersonation for certain Win32 I/O operations.
	IRequestCallbackCancel	Handles cancellation of an I/O request.
	IRequestCallbackRequest-Completion	Handles completion of an I/O request.

3.4 UMDF Driver Features

UMDF drivers can call methods in the framework and can use the Win32 API and other Windows user mode features.

Every UMDF driver must do the following:

- Support the **DllGetClassObject** export that is required by COM.
- Implement the **IClassFactory** interface to create a driver object.
- Implement the **IDriverEntry** interface on the driver class.

UMDF drivers are implemented as in-process COM servers. The COM run-time environment requires the DLL export **DllGetClassObject**, which must return an **IClassFactory** interface so that UMDF can create the driver callback object.

The **IDriverEntry** interface includes methods that initialize and uninitialize the driver and perform other required tasks when the device is added to the system. UMDF calls these methods when the driver is loaded or unloaded and when the Plug and Play manager enumerates one of the driver's devices. Every UMDF driver must implement the **IDriverEntry** interface on the driver class.

3.4.1 Impersonation

UMDF drivers run in the LocalService security context. The LocalService context has minimum security privileges on the local computer and presents anonymous credentials on the network.

When necessary, a UMDF driver can impersonate the client process to handle I/O requests, but not Plug and Play, power, or other system requests. At driver installation, the **INF** file specifies the maximum impersonation level that the driver can use.

When an application calls the **CreateFile** function, it can specify the impersonation level for the driver. The impersonation level determines the operations that the driver can perform in the context of the application process. If the level that the application specifies is different from the minimum that the **INF** provides, the reflector uses the lower of the two levels.

A driver requests impersonation by calling the **IWDFIoRequest::-Impersonate** method with the required impersonation level and a pointer to the driver's **IImpersonateCallback** interface. The **IImpersonateCallback** interface includes one method, **OnImpersonation**, which should implement the code that must be run during impersonation. To prevent security leaks, **OnImpersonation** should perform only the tasks that require impersonation and should not call any other framework methods. Properly handling impersonation is key to writing a secure driver.

3.4.2 Device Property Store

The device property store is an area in the registry where a UMDF driver can maintain information about the properties of its device. A UMDF driver can create a device property store (or retrieve an existing property store) during initialization by calling the **IWDF-DeviceInitialize::RetrieveDevicePropertyStore** method. This method returns a pointer to the **IWDFNamedPropertyStore** interface, through which the driver can set and get the values of device properties.

The format and contents of the property store is driver defined. The property store persists in the registry until the device is uninstalled from the system.

3.5 I/O Request Flow

Figure 3.2 shows the path that an I/O request takes from an application to a UMDF driver.

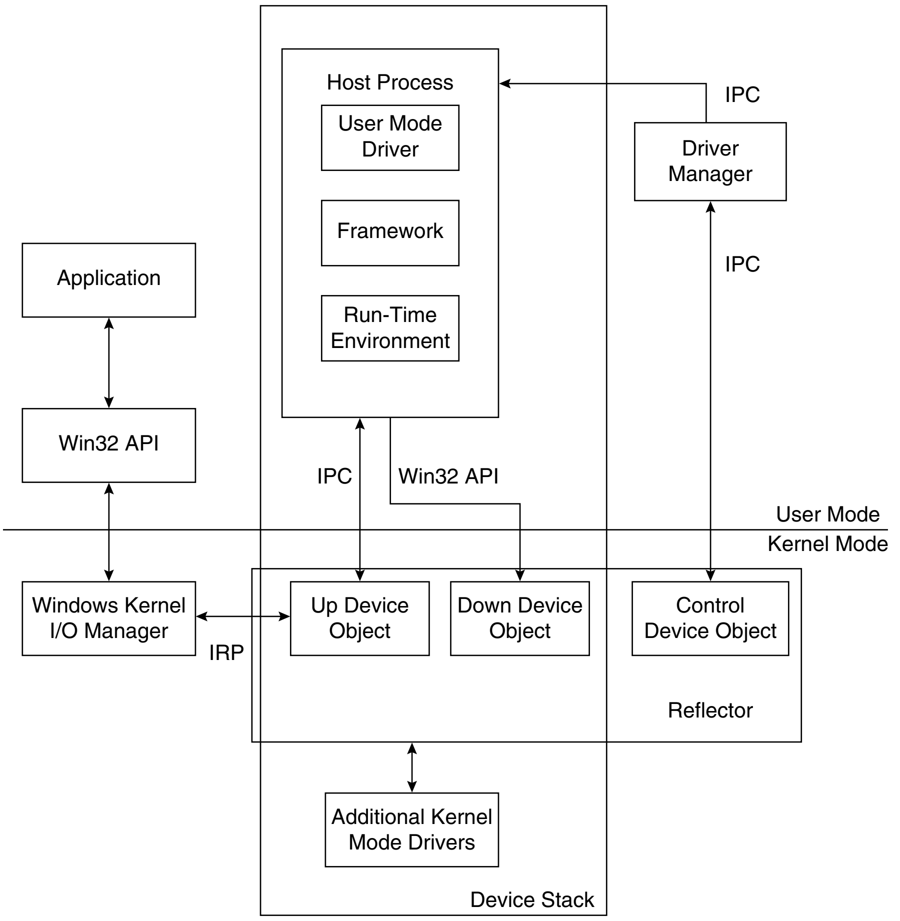

Figure 3.2 I/O Flow to UMDF Driver

Callers use the Win32 API to send I/O requests to devices that are managed by UMDF drivers, just as they do for any other device. The Win32 API calls the appropriate kernel mode I/O routine, and the Windows kernel I/O manager creates an I/O request packet (IRP) and sends this IRP to the driver at the top of the kernel mode device stack for the target device. For a device that is managed by a UMDF driver, the reflector is the driver at the top of the kernel mode device stack.

The reflector creates three types of device objects:

- up device object—This is at the top of the kernel mode stack for the device and thus is the reflector's target for IRPs from the I/O manager. When an IRP arrives, the reflector uses interprocess communications to forward it to the User Mode Driver host process, in which the framework and driver run. UMDF interprets the request and calls methods in the driver's event callback interfaces to handle it. The reflector creates an up device object for each device stack in which it participates.
- down device object—This is the reflector's target for I/O requests that originate in the User Mode Driver and is therefore the default I/O target for the bottom driver in the user mode device stack. A User Mode Driver might issue an I/O request to perform a device I/O control operation on its own device or to retrieve data from another device to complete one of its own requests. The reflector creates a down device object for each stack in which it participates.
- control device object—This manages I/O requests to and from the driver manager, which are not part of the normal I/O flow to the driver. Instead, the control device object enables "sideband" I/O between the reflector and the driver manager, which is independent of the normal flow of I/O, power management, and Plug and Play requests. The reflector creates only one control device object per system.

In addition to creating IRPs to represent user I/O requests, the I/O manager sends IRPs to notify drivers of Plug and Play requests, power management requests, change to device status, and queries about device and driver resources (among other purposes). Therefore, the UMDF I/O mode encompasses both I/O request dispatching and Plug and Play and power notification. The following sections will describe these aspects of the model in more detail.

3.5.1 I/O Request Dispatching

UMDF dispatches I/O requests to the driver, manages I/O cancellation and completion, and ensures that the Plug and Play and power state of the device is compatible with performing device I/O. Depending on the type of I/O request, the UMDF either queues the request or invokes a method in a callback interface.

UMDF provides configurable I/O queue objects that a driver can instantiate. The driver specifies which types of requests to place in each queue and how to dispatch those requests. Each queue can hold one or more types of requests.

UMDF queues and dispatches requests according to the driver's specifications: sequentially (one at a time), in parallel (as soon as they arrive), or manually (at the driver's explicit request). If Plug and Play or power management events affect queuing, a UMDF can start, stop, or resume queuing as appropriate, depending on how the driver configured the queue.

The driver provides callback interfaces to handle I/O requests on its queues. To dispatch a request, UMDF calls a method in the corresponding callback interface. For example, to dispatch a read request, UMDF calls the **OnRead** method of the driver's **IQueueCallbackRead** interface. The driver can implement request type-specific interfaces (such as **IQueue-CallbackRead**); it can also optionally implement the default I/O callback interface **IQueueCallbackDefaultIoHandler** that UMDF calls when it receives a create, read, write, or device I/O control request for which the driver has not implemented any other interface.

3.5.2 Create, Cleanup, and Close Requests

UMDF can call the driver to handle create, cleanup, and close requests or can automatically forward them to the default I/O target (typically, the next lower driver in the driver stack). The framework queues create requests if the driver configures a queue accordingly. It does not queue cleanup or close requests. Table 3.3 shows the interfaces and methods that a driver must implement to support, cleanup, and close requests.

Table 3.3 Supporting Create, Cleanup, and Close Requests

Request Type	Interface	Method
Create	**IQueueCallbackCreate**	**OnCreateFile**
Cleanup	**IFileCallbackCleanup**	**OnCleanupFile**
Close	**IFileCallbackClose**	**OnCloseFile**

Automatic forwarding is useful for drivers that process some types of I/O requests but not others. For example, a filter driver might inspect the data that is being written to a file but might not look at create, cleanup, or close requests. Therefore, it would have a callback interface for write request but would enable automatic forwarding for create, cleanup, and close.

A driver configures automatic forwarding by calling the **AutoForwardCreateCleanupClose** method on the **IWDFDevice-Initialize** interface before it creates the device object. This method sets a flag that indicates whether the framework should forward these requests. Its only parameter is one of three enumerators:

- **WdfDefault** indicates that the framework should use its defaults for forwarding. The defaults differ for filter and function drivers, as the following sections describe.
- **WdfTrue** indicates that the framework should forward requests to the default I/O target.
- **WdfFalse** indicates that the framework should not forward any create, cleanup, or close requests. If the driver does not implement the required interfaces to handle such requests, the framework fails the request.

In addition to the setting of the **AutoForwardCreateCleanupClose** flag, whether the framework dispatches, forwards, or completes create, cleanup, and close requests depends on the following:

- Whether this is a filter driver or a function driver.
- Whether the driver implements the callback interface for the request type.
- For create request only, whether the driver configures a queue for the requests.

The following sections describe what the framework does with such requests for each type of driver.

3.5.2.1 Create, Cleanup, and Close in a Filter Driver

The driver calls **IWDFDeviceInitialize::AutoForwardCreateCleanup-Close** and sets **WdfDefault**. A UMDF driver identifies itself as a filter driver by calling the **IWDFDeviceInitialize::SetFilter** method.

UMDF forwards cleanup and close requests for filter drivers to the default I/O target. If a filter driver does not implement the **IQueueCallbackCreate::OnCreateFile** method, UMDF forwards create requests, too. However, if the filter driver implements **OnCreateFile**, UMDF by default calls this method when a create request arrives. **OnCreateFile** should perform whatever filtering tasks are required and then, if appropriate, forward the request to the default I/O target.

If the filter driver sets **WdfTrue** in the call to **IWDFDeviceInitialize::AutoForwardCreateCleanupClose**, UMDF forwards create requests unless the driver implements **OnCreateFile**. If the filter driver sets **WdfFalse**, UMDF calls the corresponding method if the driver implements it; otherwise, UMDF fails the request.

If the filter driver completes a create request for a file object, it should set **AutoForwardCreateCleanupClose** to **WdfFalse** so that UMDF completes cleanup and close requests for the file object instead of forwarding them.

3.5.2.2 Create, Cleanup, and Close in a Function Driver

In a function driver, if a create request arrives for which the driver has neither implemented the **OnCreateFile** method nor configured a queue to receive create request, UMDF opens a file object to represent the device and completes the request with **S_OK**. Therefore, any function driver that does not accept create or open requests from user mode applications—and thus does not register a device interface—must implement an **IQueueCallbackCreate::OnCreateFile** method that explicitly fails such requests. Supplying a method to fail create requests ensures that a rogue application cannot gain access to the device.

To handle file cleanup and close requests, a driver implements the **IFileCallbackCleanup** and **IFileCallbackClose** interfaces. If a function driver does not implement such interfaces, UMDF closes the file object and completes the request with **S_OK**.

3.5.3 Create, Read, Write, and Device I/O Control Requests

For read, write, and device I/O control requests, the driver creates one or more queues and configures each queue to receive one or more types of I/O requests. For create requests, the driver can configure automatic forwarding, as described in the preceding section, or can direct the requests to a queue.

When such a request arrives, the I/O request handler:

- Determines whether the driver has created a queue that handles this type of request (either by explicitly configuring the queue for the request type or by creating a default I/O queue) or has implemented a default I/O handler. If neither is true, the handler fails a read, write, or device I/O control request if this is a function driver. If this is a filter driver, the handler forwards the request to the default I/O target.
- Determines whether the queue is accepting requests and the device is powered on (in the DO state). If both are true, the handler creates an I/O request object to represent the request and adds it to the queue. If the queue is not accepting requests, the handler fails the request.
- Notifies the Plug and Play handler to power up the device if the queue is power managed and the device is not in the DO state.
- Queues the request.

Figure 3.3 summarizes the flow of a create, read, write, or device I/O control request through the framework to the driver.

3.6 I/O Queues

The **IWDFIoQueue** interface exposes a queue object that presents requests from UMDF to the driver. Queues control the flow of I/O through the driver.

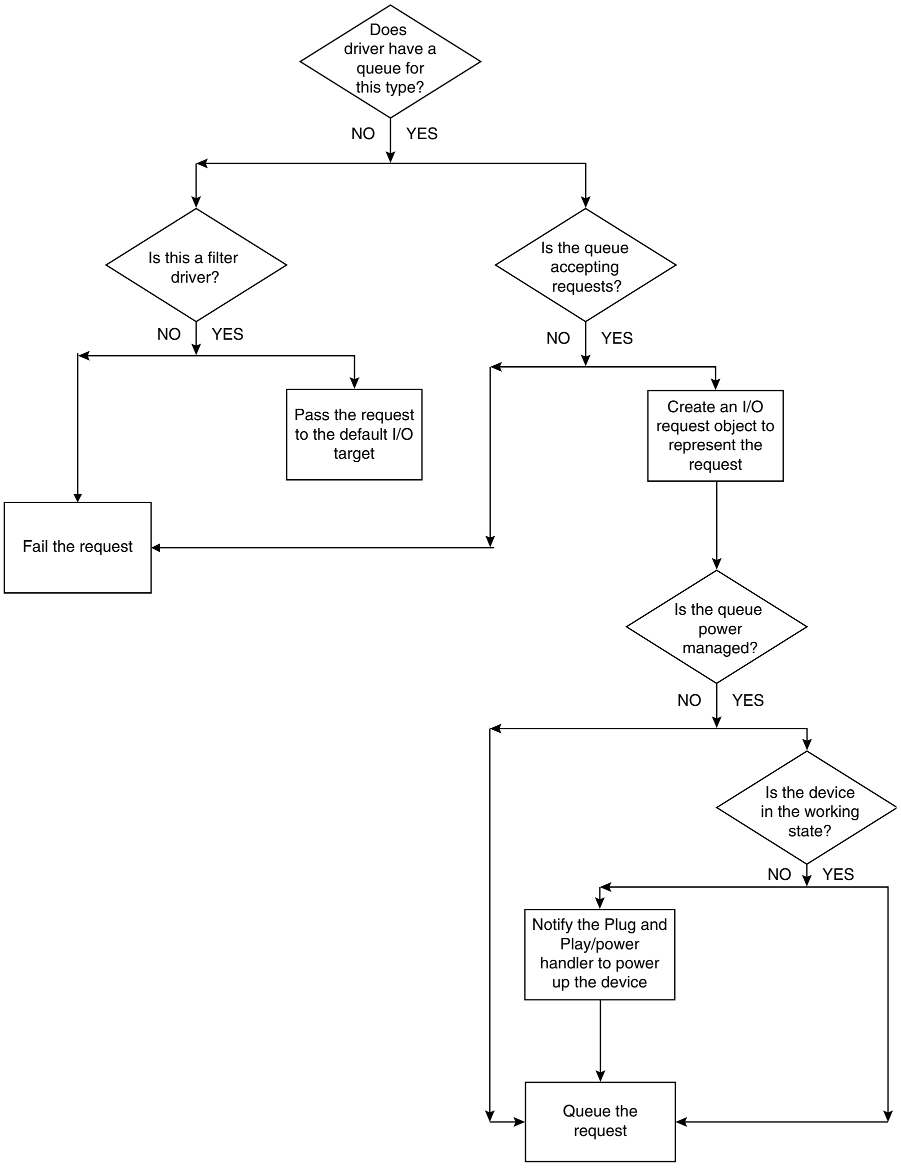
Does driver have a queue for this type?
NO
YES
Is this a filter driver?
NO
YES
Pass the request to the default I/O target
Fail the request
Is the queue accepting requests?
NO
YES
Create an I/O request object to represent the request
Is the queue power managed?
NO
YES
Is the device in the working state?
NO
YES
Notify the Plug and Play/power handler to power up the device
Queue the request

Figure 3.3 Flow of Create, Read, Write, and Device I/O Control Requests

A driver typically creates one or more I/O queues, each of which can accept one or more types of requests. The driver configures the queues when it creates them. For each queue, the driver can specify:

- The types of requests that are placed in the queue.
- The power management options for the queue.
- The dispatch method for the queue, which determines whether the framework calls the driver to dispatch a request or whether the driver calls the framework to dispatch a request. The dispatch method also determines whether the driver services multiple requests from the queue at a given time.
- Whether the queue accepts read and write requests that have a zero-length buffer.

A driver can have any number of queues, which can all be configured differently. For example, a driver might have a parallel queue for read requests and a sequential queue for write requests.

Although a request is in a queue and has not yet been presented to the driver, the queue is considered the "owner" of the request. After the request has been dispatched to the driver, it is "owned" by the driver. Internally, each queue object keeps track of which requests it owns and which requests it has dispatched to the driver. A driver can forward a request from one queue to another by calling a method on the request object.

3.6.1 Dispatch Type

A queue's dispatch type determines how and when I/O requests are delivered to the driver and, as a result, whether multiple I/O requests from a queue are active in the driver at one time. Drivers can control the concurrency of I/O requests by configuring the dispatching method for their queues. UMDF supports three dispatch types:

- **Sequential**—A queue that is configured for sequential dispatching delivers I/O requests to the driver one at a time. The queue does not deliver another request to the driver until the previous request has been completed or forwarded to another queue.
- **Parallel**—A queue that is configured for parallel dispatching delivers I/O requests to the driver as soon as possible, whether or not another request is already active in the driver.

- **Manual**—A queue that is configured for manual dispatching does not deliver I/O requests to the driver. Instead, the driver retrieves requests at its own pace by calling a method on the queue.

The dispatch type controls only the number of requests that are active within a driver at one time. It has no effect on whether the queue's I/O event callbacks are invoked sequentially or concurrently; instead, the concurrency of callbacks is controlled by the synchronization model (locking constraint) of the device object. Even if the synchronization model does not allow concurrent callbacks, a parallel queue nevertheless might have many requests active in the driver at one time.

All I/O requests that a driver receives from a queue are inherently asynchronous. The driver can complete the request within the event callback or sometime later, after returning from the callback. The driver is not required to mark the request pending, as in a Kernel Mode WDM Driver; UMDF handles this on behalf of the driver.

3.6.2 Queues and Power Management

UMDF integrates support for queues with Plug and Play/power management state machine. Power management is configurable on a per-queue basis. A driver can use both power-managed and nonpower-managed queues and can sort requests based on the requirements for its power model.

3.6.2.1 Power-Managed Queues

By default, I/O queues are power managed, which means that the state of the queue can trigger power-management activities. Such queues have a couple of advantages, as the following scenarios show:

- If an I/O request arrives while the system is in the working state (SO) but the device is not, UMDF notifies its Plug and Play and power handler so that it can restore device power.
- If the device power state begins to change while the driver "owns" an I/O request that was dispatched from a power-managed queue, UMDF can notify the driver through the **IQueueCallback-IoStop::OnIoStop** callback. The driver must complete, cancel, or acknowledge all of the I/O requests that it owns before the device can exit from the working state.

For power-managed queues, UMDF pauses the delivery of requests when the device leaves the working state (DO) and resumes delivery when the device returns to the working state. Although delivery stops while the queue is paused, queuing does not. If UMDF receives a request while the queue is paused, UMDF adds the request to the queue for delivery after the queue resumes. If an I/O request arrives while the system is transitioning to a sleep state, however, UMDF does not return the device to the working state until the system returns to the working state. The request remains in the queue until the system and the device have returned to the working state.

For requests to be delivered, both the driver and device power state must allow processing. The driver can pause delivery manually by calling **IWDFIoQueue::Stop** or **IWDFIoQueue::StopSynchronously** and later resume delivery by calling **WdfIoQueue::Start**.

3.6.2.2 Nonpower-Managed Queues

If a queue is not power managed, the state of the queue has no effect on power management, and conversely, UMDF delivers requests to the driver any time the system is in the working state, regardless of the power state of the device. Drivers should use nonpower-managed queues to hold requests that the driver can handle even while its device is not in the working state.

3.7 I/O Request Objects

The **IWDFIoRequest** interface exposes an I/O request object, which describes a read, write, or device I/O control request. When an I/O request arrives from the reflector, the I/O handler creates an I/O request object and adds the object to the queue that the driver configured for requests of that type. The driver receives a pointer to **IWDFIoRequest** interface for the object when UMDF calls the I/O event callback function or, if the queue supports manual dispatching, when the driver requests the object from the queue.

The driver can then call methods on the interface to retrieve information about the request, such as the request type, parameters, data buffers, and associated file object, among others.

Like all other UMDF objects, the I/O request object has a reference count. When the driver completes the I/O request that the object represents, UMDF automatically drops its reference on the object and any child objects such as memory buffers. After the driver that was called completes the request, it must not attempt to access the request object or any of its child objects.

3.7.1 Retrieving Buffers from I/O Requests

The **IWDFMemory** interface exposes a memory object, which encapsulates an I/O buffer that is associated with an I/O request. The memory object can be used to copy data from the driver to the buffer and vice versa. The driver can also create its own memory object by calling **IWDFDriver::CreatePreallocatedWdfMemory** and can then associate that memory object with the buffer that is supplied in an I/O request.

Like other UMDF objects, memory objects have reference counts and persist until all references to them have been removed. The buffer that underlies the memory object, however, might not be "owned" by the object itself. For example, if the issuer of the I/O request allocated the buffer or if the driver called **CreatePreallocatedWdfMemory** to assign an existing driver-created buffer to the object, the memory object does not "own the buffer." In this case, the buffer pointer becomes invalid when the associated I/O request has been completed, even if the memory object still exists.

Each memory object contains the length of the buffer that it represents. **IWDFMemory** methods that copy data to and from the buffer validate the length of every transfer to prevent buffer over runs and under runs, which can result in corrupt data or security breaches.

Each memory object also controls access to the buffer and allows the driver to write only buffers that support I/O from the device to the buffer. A buffer that is used to receive data from the device (as in a read request) is writable. The memory object does not allow write access to a buffer that only supplies data (as in a write request).

3.7.2 Sending I/O Requests to an I/O Target

If a driver cannot satisfy an I/O request by itself, it typically forwards the request to an I/O target. An I/O target represents a device object to which the driver sends an I/O request. The default I/O target is typically the next lower driver in the device stack. A UMDF driver can access the default I/O target through the **IWDFIoTarget** interface; it gets a pointer to this interface by calling the **IWDFDevice::GetDefaultIoTarget** method.

In addition to forwarding existing I/O requests, some UMDF drivers issue I/O requests by creating or reusing an I/O request object and sending the request to an I/O target. Drivers can send requests either synchronously or asynchronously and can specify a time-out value for either type of request. If the time-out period expires, UMDF cancels the request.

In addition to using the default I/O target, a driver can create additional I/O targets. An I/O target can be a UMDF driver, a KMDF driver, a WDM driver, or any other Kernel Mode Driver. UMDF defines two interfaces that create targets:

- **IWDFFileHandleTargetFactory** creates an I/O target that is associated with a file handle that the driver has already opened. The driver calls the Win32 **CreateFile** function to open the handle, and then calls methods in this interface to create the I/O target. This mechanism enables a driver to send I/O requests to a different device stack.
- **IWDFUsbTargetFactory** creates a USB device object and an associated I/O target.

To create an I/O target, the driver queries the device object for the **IWDFFileHandleTargetFactory** or **IWDFUsbTargetFactory** interface and then calls the creation method that is supported by the interface.

If the driver is the originator of the request, it creates an I/O request object by calling **IWDFDevice::CreateRequest**. If the driver is merely forwarding an existing request, this step is not required.

Whether this is a new or existing request, the driver must format it before sending it. To format a request for the default I/O target or for a file handle-based target, the driver calls methods in the **IWDFIoTarget** interface. To format an I/O request for a USB target, the driver calls methods in the **IWDFUsbTargetDevice**, which inherits from **IWDFIoTarget**. Formatting the request is important because it specifies the buffers and buffer lengths that the target should use in performing the I/O.

The driver then can call **IWDFIoRequest::Send** to send the request. If the driver implements the **IRequestCallbackCompletion::-OnCompletion** and **IRequestCallbackCancel::OnCancel** interfaces, UMDF calls the driver if the request is completed or canceled.

The I/O target object racks queued and sent requests and can cancel them when the state of the target device of the issuing driver changes. UMDF does not free the I/O target object until all of the I/O requests that have been sent to it are complete. If the driver created the I/O request, it must release its reference to the request before deleting it.

By default, UMDF sends a request only when the target is in the proper state to receive it. However, a driver can request that UMDF ignore the state of the target and send the request anyway. If the target device has been stopped (but not removed), UMDF queues the request to send later after the target device resumes. If the driver that forwarded the request specifies a time-out value, the timer starts when the request is added to the queue.

To manage an I/O target, the driver can call methods in the **IWDFIoTargetStateManagement** interface. These methods enable the driver to start, stop, and remove the target and to query its current state.

3.7.3 Creating Buffers for I/O Requests

Drivers that issue I/O requests must supply buffers with those requests. A driver can

- Allocate the buffer from memory by using the C++ **new** operator or a Win32 memory allocation function and the call **IWdfDriver::-CreatePreallocatedWdfMemory** to associate the buffer with a memory object. The driver must ensure that the buffer persists until the request has completed.
- Call **IWdfDriver::CreateWdfMemory** to create a memory object with a specified buffer size. UMDF ensures that the buffer persists until the I/O request has completed back to the issuing driver.
- Retrieve a memory object from an incoming I/O request for use in a new request.

If the driver uses a memory object, UMDF takes out a reference on that object on behalf of the new I/O target when it formats the memory object to send to the I/O target. This reference persists until one of the following occurs:

- The request has been completed.
- The driver reformats the request object by calling **IWDFRequest::-FormatUsingCurrentType** or any of the **IWDFIoTarget::Format-RequestForXxx** methods.
- The request has been deleted.

The driver can retrieve a memory object from an incoming I/O request and then reformat it for use in a new request to a new I/O target. However, if the driver has not yet completed the original request, the driver still has a reference on the memory object. The driver should implement an I/O completion callback (the **IRequestCallbackRequestCompletion** interface) for the new I/O request, and in this callback must call **Release** on the memory object before it completes the original request.

3.7.4 Canceled and Suspended Requests

Windows I/O is inherently asynchronous. The system can request that a driver stop processing an I/O request at any time for many reasons, most commonly:

- The thread or process that issued the request cancels it or exits.
- A system Plug and Play or power event such as hibernation occurs.
- The device is being, or has been, removed.

The action that a driver takes to stop processing an I/O request depends on the reason for suspension or cancellation. In general, the driver can either cancel the request or complete it with an error. In some situations, the system might request that a driver suspend (temporarily pause) processing; the system notifies the driver later when to resume processing.

To provide a good user experience, drivers should provide callbacks to handle cancellation and suspension of any I/O request that might take a long time to complete or that might not complete, such as a request for asynchronous input.

3.7.4.1 Request Cancellation

How UMDF proceeds to cancel an I/O request depends on whether the request has already been delivered to the target driver:

- If the request has never been delivered—either because UMDF has not yet queued it or because it is still in a queue—UMDF cancels or suspends it automatically without notifying the driver.
- If the request has been delivered but the driver forwards it to a different queue, UMDF automatically cancels the request without notifying the driver.

- If the request has been delivered and is owned by the driver, UMDF does not cancel it. However, if the driver explicitly marks the request cancelable by calling the **IWDFIoRequest::-MarkCancelable** method and registering a cancellation callback (**IRequestCallbackCancel::OnCancel**), UMDF notifies the driver that the request was canceled.

A driver should mark a request cancelable and register an I/O cancellation callback if either of the following is true:

- The request involves a long-term operation.
- The request might never succeed; for example, the request is waiting for synchronous input.

In the **OnCancel** callback, the driver must perform any tasks that are required to cancel the request, such as stopping any device I/O operations that are in progress and canceling any related requests that it has already forwarded to an I/O target. Eventually, the driver must complete the request with the status **ERROR_CANCELLED**.

Requests that the driver has marked cancelable cannot be forwarded to another queue. Before requeuing a request, the driver must first make it noncancelable by calling **IWDFIoRequest::UnmarkCancelable**. After the request has been added to the new queue, UMDF again considers it cancelable until that queue dispatches it to the driver.

3.7.4.2 Request Suspension

When the system transitions to a sleep state—typically because the user has requested hibernation or closed the lid on a laptop—a driver can complete, requeue, or continue to hold any requests that it is currently processing. UMDF notifies the driver of the impending power change by calling the **IQueueCallbackIoStop::OnIoStop** callback for each such request. Each call includes flags that indicate the reason for stopping the queue and whether the I/O request is currently cancelable.

Depending on the value of the flags, the driver can complete the request, requeue the request, acknowledge the event but continue to hold the request, or ignore the event if the current request will complete in a timely manner. If the queue is stopping because the device is being removed, either by an orderly removal or a surprise removal, the device must complete the request immediately.

Drivers should implement the **OnIoStop** method for any request that might take a long time to complete or that might not complete, such as a request for asynchronous input. **OnIoStop** provides a good user experience for laptops and other power-managed systems.

3.7.5 Completing I/O Requests

To complete an I/O request, a driver calls **IWDFIoRequest::Complete** or **CompleteWithInformation**. In response, UMDF completes the underlying I/O request from the system and then deletes the I/O request object and any child objects. If the driver implements the **IObjectCleanup::OnCleanup** method for the request object, UMDF invokes that method before completing the underlying system I/O request, so that the system I/O request itself is still valid when the callback runs. Because the underlying request is still valid, the UMDF driver has access to its parameters and memory buffers.

After **Complete** or **CompleteWithInformation** returns, the I/O request object and its resources have been released. The driver must not attempt to access the object or any of its resources, such as parameters and memory buffers that were passed in the request.

If the request was dispatched from a sequential queue, the driver's call to complete the request might cause UMDF to deliver the next request in the queue. (If the queue is configured for parallel dispatching, UMDF can deliver another request at any time.) If the driver holds any locks while it calls **Complete** or **CompleteWithInformation**, it must ensure that its event callback methods for the queue do not use the same locks because a deadlock might occur. In practice, this is difficult to ensure, so the best practice is not to call **Complete** or **CompleteWithInformation** while holding a lock.

3.7.6 Adaptive Time-outs

Drivers for Windows 7 should follow the Windows guidelines for I/O completion and cancellation, which require that drivers:

- Support cancellation for I/O requests that might take an indefinite period of time to complete.

- Complete I/O requests within a reasonable period (generally, 10 seconds or less) after cancellation.
- Do not block I/O thread for an unreasonable period while performing I/O. UMDF I/O threads are a limited resource, so blocking on such a thread for a long time can decrease driver performance.

To aid User Mode Drivers in conforming to these guidelines, UMDF supports adaptive time-outs. UMDF tracks the progress on critical I/O operations that can hold up the system if delayed. Critical operations include cleanup, close, cancellation, and Plug and Play and power requests.

When the reflector passes a critical request to the driver host process, it watches for progress to ensure that I/O operations are proceeding. While such a request is pending, the User Mode Driver must complete an operation at regular intervals until it has completed the critical request. If the time-out period expires, the reflector terminates the host process and reports the problem through Window Error Reporting (WER). By default, the time-out is currently one minute. If the driver must perform operations that take a long time to complete, it should handle them asynchronously, create a separate thread to handle them, or handle them in a user work item.

3.8 Self-Managed I/O

Some drivers have I/O paths that do not go through queues or are not subject to power management. UMDF provides self-managed I/O features to support these requirements.

The self-managed I/O callbacks correspond directly to Plug and Play and power management state changes, and the automatic queuing feature in UMDF is built around the same mechanism. These routines are called with a pointer to the **IWDFDevice** interface and no other parameters. If a driver implements the **IPnpCallbackSelfManagedIo** interface, UMDF calls its methods at the designated times so that the driver can perform whatever actions it requires.

Table 3.4 lists the methods of this interface and indicates when each is called.

Table 3.4 Self-Managed I/O Methods

Method	When Called
OnSelfManagedIoCleanup	During device removal, after calling **OnSelfManagedIoSuspend**
OnSelfManagedIoFlush	After device removal has completed.
OnSelfManagedIoInit	During device start-up, after the framework has called the driver's **IPnpCallback::OnDoEntry** callback function for the first time.
OnSelfManagedIoRestart	When the device returns from a low-power state to its working (DO) state; called only if UMDF previously called the driver's **OnSelfManagedIo-Suspend** method.
OnSelfManagedIoStop	Not currently called.
OnSelfManagedIoSuspend	When one of the following is true: ■ The device is about to enter a low-power state. ■ The device is being removed or was surprise-removed. ■ The Plug and Play manager is preparing to redistribute the system's hardware resources among system's attached devices.

3.9 Synchronization Issues

Because Windows is a preemptive, multitasking operating system, multiple threads can try to access shared data structures or resources concurrently and multiple driver routines can run concurrently. To ensure data integrity, all drivers must synchronize access to shared data structures.

For UMDF drivers, ensuring proper synchronization requires attention to several areas:

- The number of concurrently active requests that are dispatched from a particular queue.
- The number of concurrently active callbacks for a particular device object.
- The driver utility functions that access object-specific data.

The dispatch method for an I/O queue controls the number of requests from the queue that can concurrently be active in the driver, as described in “Dispatch Type” earlier. Limiting concurrent requests does not, however, resolve all potential synchronization issues. Concurrently active callbacks on the same object might require access to shared object-specific data. Similarly, driver utility functions might share object-specific data with callbacks. For example, a driver’s cleanup and cancellation methods often use the same data as its dispatch (read, write, and device I/O control) callbacks.

UMDF provides configurable concurrency control, called the **synchronization model** or **locking constraint**, for the callbacks of several types of objects. An object’s **synchronization model** determines whether UMDF invokes certain event callbacks on the object concurrently.

UMDF defines two synchronization models that is device scope and no scope.

Device Scope means that UMDF does not call certain I/O event callbacks concurrently for an individual device object or any file objects or queues that are children of the device object. Specifically, device scope applies to the following event callbacks:

IFileCallbackCleanup::OnCleanupFile
IFileCallbackClose::OnCloseFile
IQueueCallbackCreate::OnCreateFile
IQueueCallbackDefaultIoHandler::OnDefaultIoHandler
IQueueCallbackDeviceIoControl::OnDeviceIoControl
IQueueCallbackIoResume::OnIoResume
IQueueCallbackIoStop::OnIoStop
IQueueCallbackRead::OnRead
IQueueCallbackStateChange::OnStateChange
IQueueCallbackWrite::OnWrite
IRequestCallbackCancel::OnCancel

However, callbacks for different device objects that were created by the same driver can be called concurrently. By default, a UMDF uses device scope.

No Scope means that UMDF can call any event callback concurrently with any other event callback. The driver must create and acquire all its own locks.

A driver sets the synchronization mode by calling the **SetLocking Constraint** method of the **IWDFDeviceInitialize** interface before it creates the device object.

3.10 Locks

In addition to the synchronization for the configurable synchronization model, UMDF provides a lock for each device and I/O queue object. A driver can acquire and release this lock by using the **IWDFObject::-AcquireLock** and **IWDFObject::ReleaseLock** methods. These methods are supported for **IWDFDevice** and **IWDFIoQueue**, which inherit from **IWDFObject**.

Driver code that runs outside an event callback sometimes must synchronize with code that runs inside an event callback. After acquiring the lock, the driver can safely access the object and perform other actions that affect the object. However, to prevent a deadlock, the driver must release the lock before calling any methods in the framework, such as **IWDFRequest::CompleteWithInformation**.

3.11 Plug and Play and Power Management Notification

UMDF implements integrated Plug and Play and power management support as an internal state machine. An event is associated with the transition to each state, and a driver can supply callbacks that are invoked at specific state changes.

UMDF is designed to work with drivers on an "opt-in" basis. A UMDF driver implements Plug and Play callback interfaces for only the events that affect its device. For example, some devices require intervention immediately after they are turned on and immediately before they are turned off. The driver for such a device can implement the

IPnpCallbackHardware interface, which provides methods to be called at those times. If the device does not require such intervention, its driver does not implement the interface.

If you are familiar with WDM driver, you probably remember that any time the system power state changes, the WDM driver must determine the correct power state for its device and then issue power management requests to put the device in that state at the appropriate time. The UMDF state machine automatically handles the translation of system power events to device power events and notifies the driver to

- Transition the device to low power when the system hibernates or goes to sleep.
- Return the device to full power when the system resumes.

UMDF automatically provides for the correct behavior in device parent/child relationships. If both a parent and a child device are powered down and the child must power up, UMDF automatically returns the parent to full power and then powers up the child.

To accomplish these power transitions, a driver implements the **IPnpCallback** and **IPnpCallbackHardware** interfaces. The methods in these two interfaces are called in a defined order and each conforms to a "contract" so that both the device and the system are guaranteed to be in a particular state when the driver is called to perform an action.

In addition, requests that the framework has received and not yet delivered to the device driver can affect the power state of the device. If the driver has configured a queue for power management, the framework can automatically restore device power before it delivers the request to the driver. It can also automatically stop and start the queue in response to Plug and Play and power events.

3.12 Device Enumeration and Startup

To prepare the device for operation, UMDF calls the driver's callback routines in a fixed sequence.

The following shows the callbacks for a driver that is involved in bringing a device to the fully operations state, starting from the **DeviceArrived** state at the bottom of the following steps.

Device Operational
End Sequence

1. Enable self-managed I/O, if driver supports it.
   IPnpCallbackSelfManagedIo::OnSelfManageIoInit or **OnSelfManagedIoRestart**
2. Start power-managed queues.
   (Called only if UMDF earlier invoked **IQueueCallbackIoStop::OnIoStop** during power-down.)
   IQueueCallbackIoResume::OnIoResume
3. Notify driver of state change.
   IPnpCallback::OnD0Entry

Restart from here if device is in low-power state

4. Prepare hardware for power.
   IPnpCallbackHardware::OnPrepareHardware

Restart from here if rebalancing resources

5. Create device object.
   IDRiverEntry::OnDeviceAdd

Device Arrived
Start Sequence Above

At the bottom of the preceding steps, the device is not present on the system. When the user plugs it in, UMDF begins by calling the driver's **IDRiverEntry::OnDeviceAdd** callback so the driver can create a device object to represent the device. UMDF continues calling the driver's callback routines by progressing up through the sequences above until the device is operational. If the device was stopped because the PnP manger was rebalancing system resources or if the device was physically present but not in the working state, not all of the steps are required, as the sequence shows.

3.13 Device Power-Down and Removal

UMDF might remove a device from the operational state for several reasons:

- To put the device in a low-power state because the system is entering a sleep state

- To rebalance resources
- To remove the device after the user has requested an orderly removal
- To disable the device in response to the user's request in Device Manager

The following shows the sequence of callbacks that are involved in power-down and removal. The sequence starts at the top of the sequence below with an operational device that is in the working power state (DO).

Device Operational
Start Sequence

1. Suspend self-managed I/O.
   IPnpCallbackSelfManagedIo::OnSelfManagedIoSuspend
2. Stop power-managed queues.
   IQueueCallbackIoStop::OnIoStop
3. Notify driver of state change.
   IPnpCallback::OnDoExit

Stop here if transitioning to low-power state

4. Release hardware (Not called if target device state is **WdfPowerDeviceD3Final**).
   IPnpCallbackHardware::OnReleaseHardware

Stop here if rebalancing resources

5. Purge power-managed queues.
   IQueueCallbackIoStop::OnIoStop
6. Flush I/O buffers, if driver supports self-managed I/O.
   IPnpCallbackSelfManagedIo::OnSelfManagedIoFlush
7. Purge nonpower-managed queues.
   IQueueCallbackIoStop::OnIoStop
8. Clean up I/O buffers, if driver supports self-managed I/O.
   IPnpCallbackSelfManagedIo::OnSelfManagedIoCleanup

Device Removed
End Sequence

As the preceding sequence shows, the UMDF power-down and removal sequence involves calling the corresponding "undo" callbacks in the reverse order from which UMDF called the methods that it invoked to make the device operational.

3.13.1 Surprise-Removal Sequence

If the user removes the device without warning, by simply unplugging it without using Device Manager or the Safely Remove Hardware utility, the device is considered *surprise removed*. When this occurs, UMDF follows a slightly different removal sequence. It also follows the surprise-removal sequence if another driver invalidates the device state (for example, a Kernel Mode Driver calls **IoInvalidateDeviceState**), even if the device is physically present.

In the surprise-removal sequence, UMDF calls the **IPnpCallback::-OnSurpriseRemoval** callback to notify the driver that the device has been unexpectedly removed. This callback is not guaranteed to occur in any particular order with the other callbacks in the removal sequence.

Drivers for all removable devices must ensure that the callbacks in both the shutdown and startup paths can handle failure, particularly failures that are caused by the removal of the hardware. The reflector times out the driver if an attempt to access the hardware waits indefinitely.

The following sequence shows the surprise-removal sequence.

Device Surprise Removed

Start here if device is in the working state

1. Suspend self-managed I/O.
   IPnpCallbackSelfManagedIo::OnSelfManagedIoSuspend
2. Stop power-managed queues.
   IQueueCallbackIoStop::OnIoStop
3. Notify driver of state change.
   IPnpCallback::OnDOExit

Start here if device is not in the working state

4. Release hardware.
   IPnpCallbackHardware::OnReleaseHardware
5. Purge power-managed queues.
   IQueueCallbackIoStop::OnIoStop

6. Flush I/O buffers, if driver supports self-managed I/O.
   IPnpCallbackSelfManagedIo::OnSelfManagedIoFlush
7. Purge nonpower-managed queues.
   IQueueCallbackIoStop::OnIoStop
8. Clean up I/O buffers, if driver supports self-managed I/O.
   IPnpCallbackSelfManagedIo::OnSelfManagedIoCleanup

Removal Processing Complete

If the device was not in the working state when it was removed, UMDF starts with the **IPnpCallbackHardware::OnReleaseHardware** method. It omits the intervening steps, which were already performed when the device exited from the working state.

3.14 Build, Test, and Debug

Like Kernel Mode Drivers, UMDF drivers are built with the Windows Driver Kit (WDK). The WDK includes the required libraries and header files to build a UMDF driver, along with sample code and build scripts.

The WDK includes a library of debugger extensions for use with UMDF drivers. The debugger extensions work with CDB, NTSD, and WinDbg. UMDF drivers can be debugged in either user mode or kernel mode. Debugging a User Mode Driver is similar to debugging a service. Symbol files are also included as an aid in debugging.

UMDF drivers can use Event Tracing for Windows (ETW) to generate a trace log of driver events. The logs can be viewed using the tracing tools provided with the WDK.

The user mode components that Microsoft supplies start trace sessions that record their activities and note such events as driver hangs, time-outs, and failures. UMDF has integrated verification code that is always enabled. If a driver uses UMDF DDIs incorrectly or passes incorrect parameters, the verifier saves a memory dump in the system log file directory and optionally creates a Windows error report. The trace log files and error reports can optionally be sent to the Microsoft Windows Error Reporting (WER) facility. WER captures software crash data and supports end-user reporting of crash information. The information collected through WER is accessible to vendors so that they can analyze problems and respond directly to their customers.

You can also use PREfast, a static analysis tool that is part of the WDK, with UMDF drivers. PREfast examines code paths on a per-function basis and can find logic and usage errors in a driver at compile time.

3.14.1 Installation and Configuration

UMDF drivers are installed by using standard **INF** files, in the same way as Kernel Mode Drivers. A hardware vendor's installation package thus includes

- An **INF** file for the driver
- The redistributable update co-installer WUDF_Update_*MMmmm*.dll (where *MM* is the major UMDF version number and *mmm* is the minor version number)
- A driver binary
- An optional custom installation application

The **INF** file includes a [DDInstall.**CoInstallers**] section that references the co-installer. The co-installer is available to hardware vendors in the WDK and in general distribution releases (GDRs). Microsoft provides the redistributable co-installer Wudf_Update_MMmmm.dll, which vendors can ship as part of their driver packages. The co-installer is a signed component. Driver installation fails if the certificate with the co-installer was signed and is not available on the target system.

The **Wdf** section of the driver's **INF** file specifies the driver service name, the driver class ID, the order in which the UMDF drivers should be installed in the device stack, and the maximum impersonation level that the UMDF driver can use.

The **DestinationDirs** section of the **INF** file specifies the location to which to copy the driver. All UMDF drivers must reside in %SystemRoot%\system32\drivers\UMDF. This directory has sufficient ACLs to ensure that an unprivileged user cannot tamper with the drivers.

If the UMDF device stack for the driver contains only filter drivers, the **INF** installs the reflector as the top-level filter driver in the Kernel Mode Driver stack. However, if the function driver for the device is a UMDF driver, the **INF** installs the reflector as the service for the device.

When the **INF** for the device is processed, the co-installer starts the driver manager if it is not already running. User Mode Driver installation must not require a reboot of the system.

3.14.2 Versioning and Updates

UMDF supports versioning, which enables a driver binary to run with the same major version of UMDF independent of the operating system version. Versioning helps to ensure that a driver uses the components with which it was designed, tested, and released. Each version of the operating system supports all versions of UMDF. Updates to UMDF or the operating system do not require updates to the driver. Microsoft is responsible for ensuring consistent behavior across releases.

Although two major versions of UMDF can run side by side simultaneously, two minor versions of the same major version cannot. At installation, a more recent minor version of the UMDF run-time library or other components overwrites an existing, older minor version. If the older version is already loaded when a user attempts to install a driver with a newer version, the user must reboot the system.

This page intentionally left blank

CHAPTER 4

PROGRAMMING DRIVERS FOR THE USER MODE DRIVER FRAMEWORK

In this chapter, we will cover the overview of the User Mode Driver Framework (UMDF) as well as how to create a UMDF Driver. We will introduce COM and how it is used in conjunction with creating a UMDF Driver. As was discussed in Chapter 3, UMDF Drivers create callback objects to represent the driver itself, each of its devices, and each of its I/O queues. The framework defines a set of interfaces, some of which the driver must implement for the callback objects that it creates and others the framework itself implements for the objects that the framework creates. In general, the framework creates an object that corresponds to each of the driver's callback objects. The framework makes the interfaces on its own objects available to the driver by passing pointers.

4.1 Windows I/O Overview

Before writing a driver—even a User Mode Driver—you must understand the basics of the Windows I/O architecture, as shown in Figure 4.1. In this overview, we will discuss some of the major principles behind the I/O system and define the most important terms. For a detailed explanation of Window I/O, see the book *Windows Internals, Fifth Edition*, which is listed in the Bibliography.

As Figure 4.1 shows, Windows supports a layered I/O architecture. User mode applications and services issue I/O requests through the Win32 API and communicate with the user mode PnP manager to perform Plug and Play and power activities. The Win32 API and user mode PnP manager,

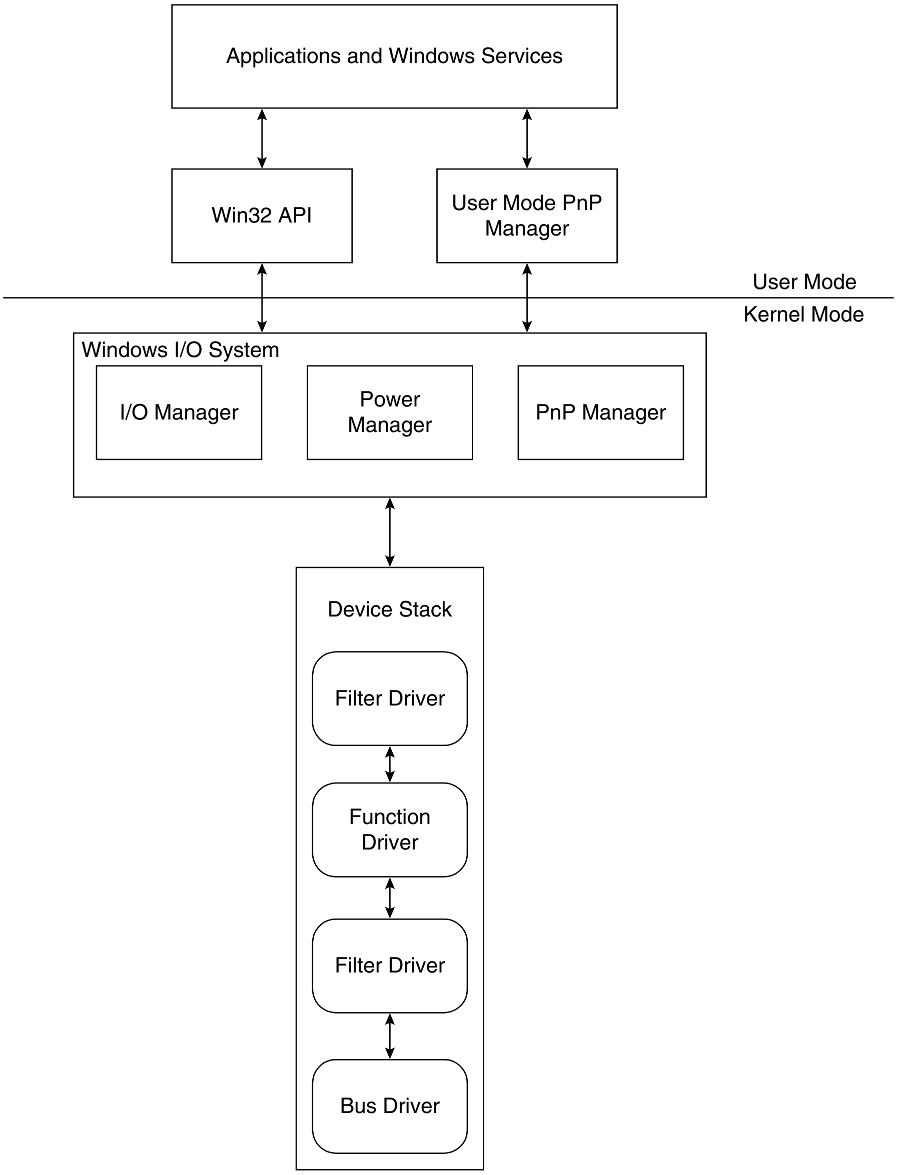

Figure 4.1 Block Diagram Windows I/O

in turn, communicate with the kernel mode I/O system, which includes the kernel mode I/O manager, PnP manager, and power manager. The kernel mode I/O system communicates with the Kernel Mode Drivers.

Drivers are also layered. Most devices are driven by not one but by several drivers, each layered atop the next. Together, the group of drivers that operates a particular device is called the device stack (sometimes also called the driver stack). At the bottom of the device stack is a bus driver, which controls a bus and enumerates the devices that are connected to the bus. Layered above the bus driver are filter drivers and a function driver. The function driver is the primary driver for the device and exposes the device interface to the I/O manager. Filter drivers can be layered above or below the function driver and provide additional features, such as encryption or security, or change the behavior of a device. Each driver in the device stack is represented by a device object. The device object is a data structure that contains information about the driver and the device.

Drivers receive I/O, Plug and Play, and power management requests in the form of I/O request packets (IRPs). When the Windows I/O manager receives an I/O request, it determines which device stack corresponds to the virtual file that is specified in the request. It then packages the request into an IRP and forwards it to the target device object. Plug and Play and power management notifications are also packaged as IRPs, and drivers communicate with other drivers by sending IRPs.

When a driver receives an IRP, it takes whatever actions are required to satisfy the request and then completes it. Sometimes, however, a driver cannot satisfy an IRP by itself. If the driver cannot complete the IRP, it typically passes the IRP down the device stack to the next driver and optionally sets an I/O completion routine for callback when the request is complete. Eventually, the IRP arrives at a driver that satisfies and completes the request. When the request is complete, the I/O manager calls any completion callback routines that drivers set as the request traveled down the device stack. It calls these routines in the opposite order in which they were set—that is, it "unwinds" back up the device stack.

4.2 Brief COM Information

Chapter 5 covers the component object model (COM) in a fair amount of detail to get you up to speed on COM and the UMDF driver development. Here, we will just touch upon a few points.

The UMDF interfaces are defined in terms of COM. UMDF does not depend on the COM infrastructure and run-time library. Instead, it uses only the COM programming pattern, specifically the query-interface and reference-counting features. It does not use the COM run-time loader.

The following is a brief overview on COM:

- COM is based on a client-server model.
- COM maintains reference counts for all of its objects.
- COM objects expose interfaces, which support callable methods.
- COM interfaces are C++ abstract base classes. An interface contains one or more methods that form the contract for any caller that wants to use the class.
- The query-interface (QI) feature of COM enables a client to query a server to determine whether the server supports a particular interface. UMDF drivers can request notification of particular system events by exposing callback interfaces. UMDF uses the QI feature to discover these callback interfaces.
- **IUnknown** is the fundamental COM interface, and every COM object supports it. **IUnknown** supports the **QueryInterface**, **AddRef**, and **Release** methods. The **QueryInterface** method enables other components to determine which interfaces the object supports. The **AddRef** and **Release** methods manage object lifetime.
- The **IClassFactory** interface creates instances of class objects. UMDF calls **DllGetClassObject** to get a pointer to an **IClassFactory** interface in the driver and then uses the **CreateInstance** method of the **IClassFactory** interface to create an instance of the driver object.
- When COM returns an interface pointer to a driver, it takes out a reference on the corresponding object. The driver should release this reference by calling the object's **Release** method when it has finished using the object. Failing to release references causes object leaks, which consume memory unnecessarily.

4.3 UMDF Architecture

A UMDF driver is a dynamic-link library (DLL) that runs as an in-process COM server. Figure 4.2 shows the components that are involved when a UMDF driver controls a device.

As Figure 4.2 shows, the User Mode Driver runs under the Host Process, which combines with Kernel Mode Drivers (including the reflector) to form the device stack for the device.

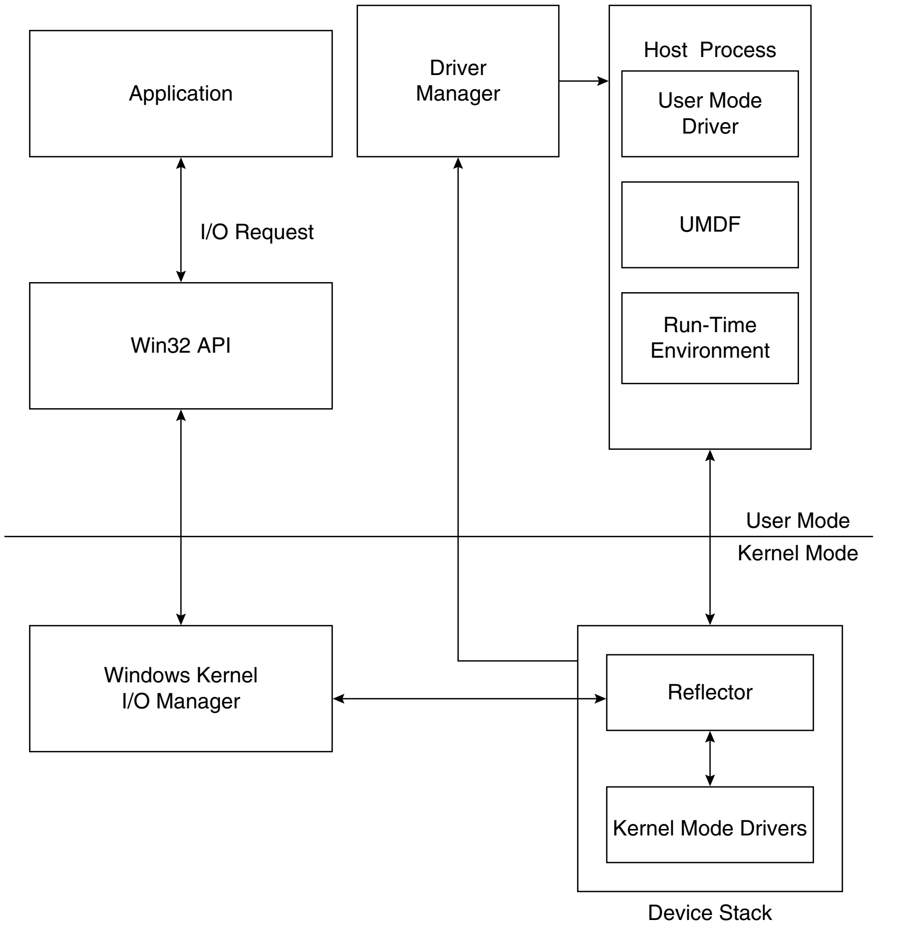

Figure 4.2 User Mode Driver Architecture

The following describes the preceding components in the figure according to the typical flow of an I/O request.

Application. The application issues I/O requests through the Win32 API, which in turn calls I/O routines in the Windows kernel.

Windows kernel. The Windows kernel creates IRPs to represent the requests and forwards them to the top of the kernel mode device stack for the target device.

Reflector. The reflector is a Kernel Mode WDM Filter Driver that is installed at the top of the kernel mode device stack for each device that is managed by a UMDF driver. The reflector manages communication between the kernel mode components and the User Mode Driver host process.

Driver manager. The driver manager creates and shuts down all the driver host processes and maintains status information about them. It also responds to messages from the reflector. The driver manager runs as a Windows service and is started during installation of the first device that is managed by a UMDF driver. The driver manager must be running all the time that any device controlled by a UMDF driver is installed on the system. Microsoft provides the driver manager.

Host process. The host process is the process in which the User Mode Driver runs. The host process is a child process of the driver manager and runs in the security credentials of a LocalService account, although it is not a Windows service. The host process includes the following components:

- The UMDF driver is an in-process COM component that controls the hardware from user mode.
- UMDF exposes the User Mode device-driver interface (DDI). UMDF is a DLL of COM-style objects that support the presentation, flow, and management of I/O and Plug and Play requests to the driver.
- The run-time environment dispatches I/O requests, loads the driver, constructs and destroys the user mode device stack, manages a user mode thread pool, and handles messages from the reflector and the driver managers.

Kernel Mode Drivers. Additional Kernel Mode Drivers can service each device.

4.4 Required Driver Functionality

Every UMDF driver must provide the following functionality:

- Support the **DllMain** export as its primary entry point.

  The driver must support **DllMain** as its primary entry point. After the system creates the UMDF driver host process, the host process loads the driver by calling the **DllMain** function. In a UMDF driver, this function does very little; typically, it enables tracing and then returns. As in all DLLs, the driver must not make blocking calls such as **WaitForSingleObject**, which can deadlock the system. Drivers should defer resource allocation to **IDriverEntry::-OnInitialize**, instead of **DllMain**.

- Support the **DllGetClassObject** export, which must return a pointer to an **IClassFactory** interface that creates an instance of the driver callback object.

  The driver must also support the **DllGetClassObject** function, which COM requires. This function returns a pointer to an **IClassFactory** interface with which UMDF can create an instance of the driver callback object. The UMDF sample drivers show how to implement this function. Alternatively, the Active Template Library (ATL) wizard can be used to generate the supporting COM code.
- Implement the **IDriverEntry** interface on the driver class.

  Finally, every User Mode WDF driver must implement the **IDriverEntry** interface on the driver class. This interface includes methods that initialize and uninitialize driver-wide data. UMDF calls the **OnInitialize** method when the first device for the driver is loaded and calls the **OnDeinitialize** method when the driver is unloaded. **IDriverEntry** also includes the **OnAddDevice** method, which UMDF calls when the Plug and Play manager enumerates one of the driver's devices.

Figure 4.3 shows the interactions of the functionality of a driver with UMDF and the system.

When the system starts, the driver manager is loaded and the following actions occur:

1. The driver manager creates the driver host process and then loads the driver library by calling the **DllMain** entry point. **DllMain** performs any required global initialization for the driver, such as starting tracing.
2. UMDF creates a framework driver object and calls the driver at **DllGetClassObject** to get an interface that I can use to create a corresponding callback object in the driver. **DllGetClassObject** returns a pointer to the driver's **IClassFactory** interface.

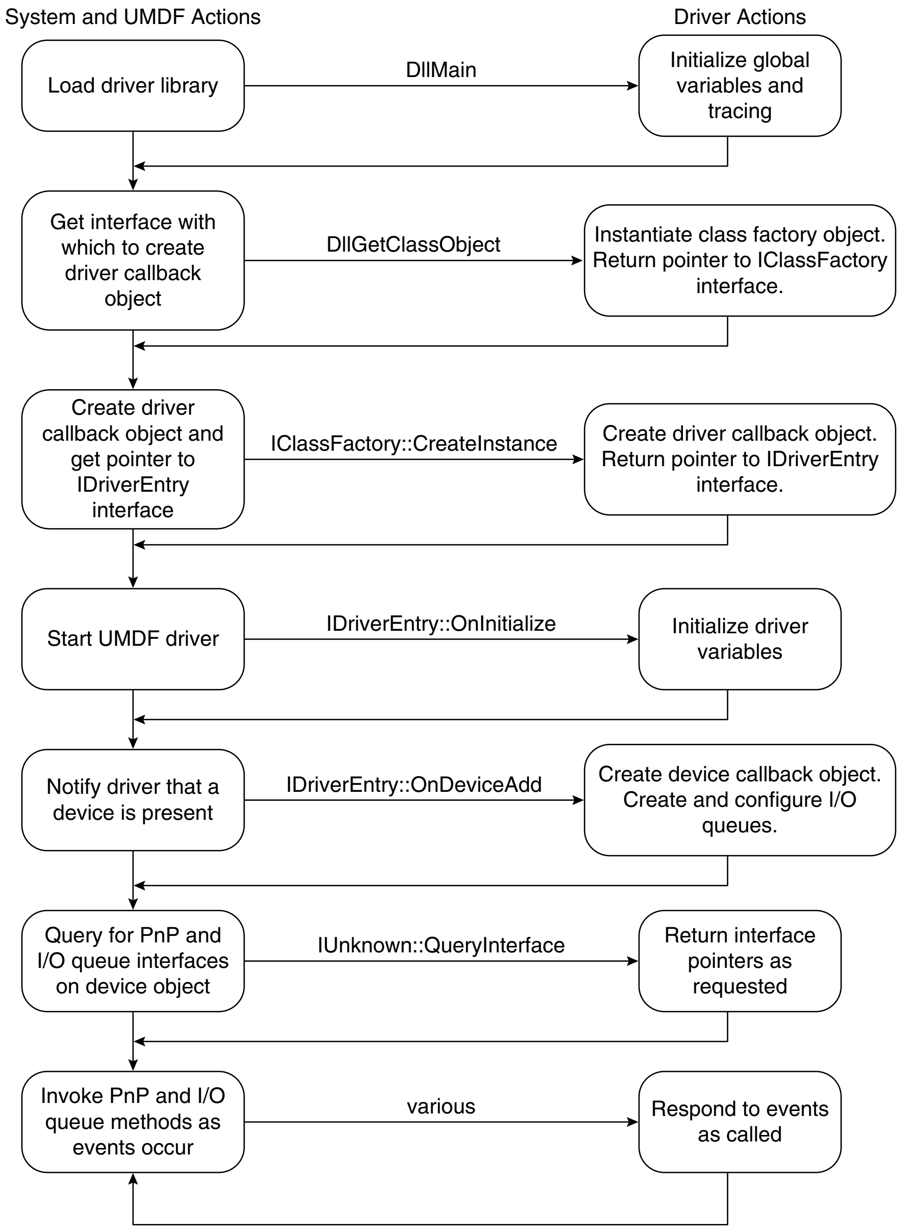
System and UMDF Actions
Driver Actions
Load driver library
DllMain
Initialize global variables and tracing
Get interface with which to create driver callback object
DllGetClassObject
Instantiate class factory object. Return pointer to IClassFactory interface.
Create driver callback object and get pointer to IDriverEntry interface
IClassFactory::CreateInstance
Create driver callback object. Return pointer to IDriverEntry interface.
Start UMDF driver
IDriverEntry::OnInitialize
Initialize driver variables
Notify driver that a device is present
IDriverEntry::OnDeviceAdd
Create device callback object. Create and configure I/O queues.
Query for PnP and I/O queue interfaces on device object
IUnknown::QueryInterface
Return interface pointers as requested
Invoke PnP and I/O queue methods as events occur
various
Respond to events as called

Figure 4.3 Flow of Control for UMDF and Driver

3. UMDF calls the **CreateInstance** method of the **IClassFactory** interface to create an instance of the driver callback object. The driver callback object implements methods to initialize the driver, to notify it that one of its devices has been enumerated, and to prepare it for unloading.
4. UMDF calls the **OnInitialize** method of the driver callback object to initialize the driver.
5. Whenever one of the driver's devices is enumerated, UMDF calls **OnDeviceAdd**. **OnDeviceAdd** performs any required configuration, creates a device callback object to represent the device, creates any required device interfaces, and creates and configures the queues into which UMDF will place I/O requests that are targeted at the driver. A device interface is a device feature that driver exposes to applications or other system components, whereas a COM interface is a related group of functions that act on an object.
6. UMDF queries for the Plug and Play and queue interfaces that it will use to handle I/O requests.
7. When the device is removed, UMDF calls the Plug and Play methods that are appropriate for the type of removal (orderly or surprise), deletes the objects, and calls the **IDriverEntry::OnDeinitialize** method to clean up. It then unloads the DLL and deletes the driver host process.

4.5 UMDF Sample Drivers

The UMDF release for Windows 7 includes several sample User Mode Drivers, which are installed at WDF\UMDF\src in the Windows Driver Kit (WDK) installation directory. You can use these samples as the basis for your own driver, as well as the examples in the book. Table 4.1 lists the UMDF samples.

Use the WDK build environment to build the samples. To build a particular sample:

1. Start a build environment window.
2. Set the working directory to the directory that contains the sample to build.
3. Type the following command: **build –ceZ**

Table 4.1 List of UMDF Sample Drivers

Name	Description
Skeleton	Minimal software-only driver that shows the structure and fundamental components of a simple UMDF driver.
Echo	Simple driver that uses a serial I/O queue and handles one I/O request at a time. The Echo sample defers the completion of each I/O request to a worker thread and shows how to mark a request cancelable while it is pending in the driver. This sample was adapted from the KMDF Echo sample and is functionally similar to the WDM sample of the same name.
USB\Driver	Hardware driver for the USB-FX2 Learning Kit from OSR. This sample supports a parallel I/O queue. It is similar to the Echo sample, but can handle multiple independent requests at one time. This driver also demonstrates how a User Mode Driver controls a device. It uses the memory in the OSR-USBFx2 device as a buffer and uses the WinUSB API and WinUSB Kernel Mode Driver to control the hardware. The driver demonstrates how to escape from UMDF and send I/O by using an alternate path and how to synchronize I/O on the alternate path with cancellation and file closure.
USB\Filter	Filter driver for the WinUSB driver stack. This sample modifies the data in read and write requests as they flow through the device stack and uses I/O targets to communicate with a lower driver.

4.5.1 Minimal UMDF Driver: The Skeleton Driver

The Skeleton driver contains the minimum amount of code that is required in a loadable UMDF driver. It was designed as a starting point from which to build drivers for actual hardware. In addition to demonstrating the minimal required features and best practices, the Skeleton driver splits into appropriate modules the common code that is required in all UMDF drivers.

The Skeleton driver supports a driver entry point, functions to create and initialize the driver and device callback objects, and functions for COM support. It does not support any I/O or Plug and Play operations. Table 4.2 lists the component source files.

Table 4.2 Component Source Files

Filename	Description
Comsup.cpp	Source code for the **CUnknown** and **CClassFactory** classes.
Comsup.h	Header file for COM support functions and classes.
Device.cpp	Source code for the device callback object class, **CMyDevice**.
Device.h	Header file for the device callback object.
Dllsup.cpp	Source code for the driver entry point and exported COM support functions.
Driver.cpp	Source code for the driver callback object class, **CMyDriver**.
Driver.h	Header file for the driver callback object.
Exports.def	Definition file that identifies the library name and exported entry point for driver.
Internal.h	Header file for local type definitions.
Makefile	Generic makefile for building the sample.
Makefile.inc	Additional commands input to the makefile.
Skeleton.htm	Help file that describes the sample.
Skeleton.rc	Resource file for the sample.
Sources	Source file for the build procedure.
UMDF_Skeleton_OSR.inx	**INF** that installs the Skeleton sample as a driver for the OSR USBFX2 device.
UMDF_Skeleton_OSR_xp.inx	**INF** that installs the Skeleton sample as a driver for the OSR USBFX2 device on Windows 7.
UMDF_Skeleton_root.inx	**INF** that installs the Skeleton sample as a driver for a root-enumerated device.
UMDF_Skeleton_root_xp.inx	**INF** that installs the Skeleton sample as a driver for a root-enumerated device on Windows 7.

4.5.2 Skeleton Driver Classes, Objects, and Interfaces

The Skeleton driver implements the four classes that are listed in Table 4.3.

Table 4.3 Classes Implemented in Skeleton Driver

Class Name	Description	Public Interfaces
CUnknown	Base Class from which others derive	**IUnknown**
CClassFactory	Class factory that instantiates the driver class	**IClassFactory**
CMyDriver	Driver callback object class	**IDriverEntry** **IUnknown**
CMyDevice	Device callback object	**IUnknown**

Figure 4.4 shows how the objects that are instantiated from these classes interact with the corresponding framework objects.

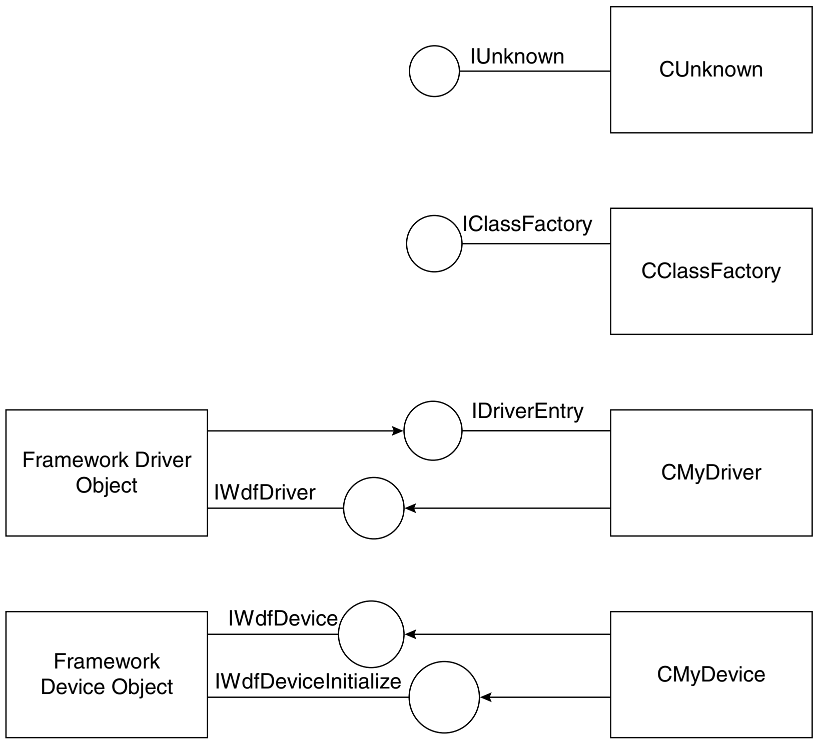

Figure 4.4 Interaction of Framework and Skeleton Objects

As Figure 4.4 shows, the framework implements a driver object and a device object. The framework's driver object uses the **IDriverEntry** interface on the Skeleton driver's **CMyDriver** object, and the **CMyDriver** object, in turn, uses the **IWdfDriver** interface on the framework's driver object. The framework's device object exposes the **IWdfDevice** and **IWdfDeviceInitialize** interfaces, which the driver's **CMyDevice** object uses. The Skeleton driver's **CMyDevice** class does not implement additional interfaces because it does not support hardware or handle I/O requests. The device object in a typical driver would implement additional interfaces for Plug and Play notifications, I/O requests, I/O queues, and so forth.

4.6 Driver Dynamic-Link Library and Exports

Every UMDF driver must support **DllMain** as the driver's primary entry point and must export the **DllGetClassObject** function so that COM can instantiate the driver object. The Skeleton sample defines these functions in the **Dllsup.cpp** file.

4.6.1 Driver Entry Point: DllMain

DllMain is the driver's primary entry point. This function typically initializes any data that is required for tracing and debugging but otherwise does little because most driver-and-device-specific initialization takes place in conjunction with the driver and device object creation. In the Skeleton driver, **DllMain** simply initializes tracing, as the following source code shows:

```
BOOL
WINAPI
DllMain{
   HINSTANCE ModuleHandle,
   DWORD Reason,
   PVOID     /* Reserved */
   }
/*++

   Routine Description:

   This is the entry point and exit point for the I/O
   driver. It does very little because the driver has mini-
   mal global data.
```

```
    This method initializes tracing.

    Arguments:
    ModuleHandle - the Dll handle for this module.
    Reason - the reason this entry point was called.
    Reserved - unused

    Return Value:
    TRUE
--*/
{
    If (DLL_PROCESS_ATTACH == Reason)
    {
        //
        // Initialize tracing.
        //

        WPP_INIT_TRACING(MYDRIVER_TRACING_ID);

    }
    else if (DLL_PROCESS_DETACH == Reason)
    {
        //
        // Clean up tracing.
        //

        WPP_CLEANUP();
    }

    return TRUE;
}
```

When the driver host process calls **DllMain**, it passes a reason for the call, along with a handle and a reserved value, both of which the function can ignore. If the driver host process is starting and the **DLL** is being loaded, the reason is **DLL_PROCESS_ATTACH**. In this case, the function initializes tracing. If the driver host process is terminating or the library did not load successfully, the reason is **DLL_PROCESS_DETACH**, so the function ends tracing. Starting and ending tracing in **DllMain** ensures that trace information is recorded for the entire life of the driver.

4.6.2 Get Class Object: DllGetClassObject

COM calls the driver's **DllGetClassObject** function to get a pointer to an interface through which it can instantiate a driver callback object. This method should create an instance of the class factory for the driver object; UMDF later calls methods on the class factory to actually instantiate the driver callback object.

The following is the source code for the Skeleton driver's **DllGet-ClassObject** function:

```
HRESULT
STDAPICALLTYPE
DllGetClassObject{
    __in REFCLSID ClassId,
    __in REFIID InterfaceId,
    __deref_out LPVOID *Interface
    }
/*++
    Routine Description:
    This routine is called by COM to instantiate the skeleton
    driver callback object and do an initial query interface
    on it.

    This method only creates an instance of the driver's
    class factory, which is the minimum required to support
    UMDF.

    Arguments:
    ClassId - the CLSID of the object being "gotten"
    InterfaceId - the interface the caller wants from that
    object
    Interface - a location to store the referenced interface
    pointer

    Return Value:
    S_OK if the function succeeds, or
    Error code indicating the reason for failure

--*/
{
    PCClassFactory factory;

    HRESULT hr = S_OK;

    *interface = NULL;
    //
    // If the CLSID doesn't match that of our "coclass"
    // defined in the IDL file) then we can't create the
    // object that the caller wants. This error may indicate
    // that the COM registration is incorrect, and another
    // CLSID is referencing this driver.
    //
    If (IsEqualCLSID(ClassId, CLSID_MyDriverCoClass == false)
    {
       Trace{
          TRACE_LEVEL_ERROR,
          L"ERROR: Called to create instance of unrecognized
            class
          "(%!GUID!)",
          };

       Return CLASS_E_CLASSNOTAVAILABLE;
    }

    //
    // Create an instance of the class factory for the caller
    //

    factory = new CClassFactory();

    if (NULL == factory)
    {
       Hr = E_OUTOFMEMORY;
    }

    //
    // Query the object we created for the interface that the
    // caller requested. Then release the object. If the QI
    // succeeded and referenced the object, its reference
    // count will now be 1. If the QI failed, the reference
    // count is 0 and the object is automatically deleted.
    //

    If (S_OK == hr)
    {
       hr = factory->QueryInterface(InterfaceId, Interface);
       factory->Release();
    }

    return hr;
}
```

In the Skeleton driver, **DllGetClassObject** verifies that the class ID passed in by the caller (COM) matches the class ID of the object, as defined in the IDL file. It creates an instance of the class factory, calls **QueryInterface** on the class factory to get a pointer to the **IClassFactory** interface and increment its reference count, and then returns.

4.7 Functions for COM Support

The source file **Comsup.cpp** supplies code that is required to support COM. It implements methods for the **IUnknown** and **IClassFactory** interfaces. This section briefly describes what these methods do, but does not show any of the sample code. You can simply copy the **Comsup.cpp** and **Comsup.h** files for use in your own drivers, typically without any changes.

4.7.1 IUnknown Methods

CUnknown is the base class from which all other classes derive, and it supports the **IUnknown** interface. Every UMDF driver must implement this class with a constructor method and the **IUnknown** interface, which includes the **AddRef**, **QueryInterface**, **QueryIUnknown**, and **Release** methods. Table 4.4 summarizes the **IUnknown** methods.

Table 4.4 IUnknown Method Names

Method Name	Description
CUnknown	Constructor, which initializes the reference count for this instance of the **CUnknown** class to 1.
QueryInterface	Returns a pointer to the **IUnknown** interface for the object.
QueryIUnknown	Public helper method that casts a **CUnknown** pointer to an **IUnknown** pointer.
AddRef	Increments the reference count for the object.
Release	Decrements the object's reference count and deletes the object if the reference count reaches zero.

4.7.2 IClassFactory Interface

The **CClassFactory** class object implements the **IClassFactory** interface. The framework invokes methods in this interface to create an instance of the driver callback class. The driver callback class instance is the only callback object that the framework creates; the driver itself creates all other callback objects in response to calls from the framework. Table 4.5 summarizes the methods in this interface.

4.7.3 Driver CallBack Object

When UMDF gets a pointer to the **IClassFactory** interface, it calls the **CreateInstance** method in that interface to create an instance of an object. That method, in turn, calls the **CMyDriver::CreateInstance** method, which creates and initializes the driver callback object. In general, any **CreateInstance** method is a factory method that creates an object.

Table 4.5 IClassFactory Methods

Method	Description
QueryInterface	Returns a pointer to the requested interface.
QueryIClassFactory	Public helper method that casts a **CClassFactory** pointer to an **IClassFactory** pointer; essentially similar to the **IQueryIUnknown** method in the **IUnknown** interface.
CreateInstance	Creates an instance of the driver callback class and returns a pointer to a requested interface for that class.
LockServer	Maintains a lock count that indicates whether the driver DLL should remain in memory.

CMyDriver::CreateInstance is defined in the source file **Driver.cpp** and is straightforward, as the following shows:

```
HRESULT
CMyDriver::CreateInstance(
   __out PCMyDriver *Driver
)
/*++

   Routine Description:

This static method is invoked to create and initialize a new
instance of the driver class. The caller should arrange for
the object to be released when it is no longer in use.

   Arguments:

      Driver - a location to store a referenced pointer to
      the new instance

   Return Value:

      S_OK if successful, or error otherwise

--*/
{
PCMyDriver driver;
HRESULT hr;

//
// Allocate the callback object
//
driver = new CMyDriver();

if (NULL == driver)
{
return E_OUTOFMEMORY;
   }

   //
   // Initialize the callback object
   //
   hr = driver->Initialize();
if (SUCCEEDED (hr))
{
      //
      // Store a pointer to the new, initialized object in
      // the output parameter.
      //
      *Driver = driver;
   }
   else
   {

      //
      // Release the reference on the driver object so that
      // it will delete itself
      //
      driver->Release();
}
return hr;
}
```

This method allocates and creates an instance of the driver callback object, and then calls the initialize method to initialize the object. The **Skeleton** driver object requires no initialization, so the **Initialize** method is a stub (and so is not shown here). **CreateInstance** returns a pointer to the new driver callback object and releases its reference on this object before returning.

Every UMDF driver must implement the **IDriverEntry** interface on the driver callback object. This interface supports methods to initialize the driver, perform tasks when one of the driver's devices is added to the system, and prepare the driver for unloading, just before **DllMain.DetachProcess** is called. The **driver.cpp** file contains code that implements **IDriverEntry**.

IDriverEntry defines three methods: **OnDeviceAdd, OnInitialize**, and **OnDeInitialize**. In the **Skeleton** driver, the **OnInitialize** and **OnDeInitialize** methods are stubs.

When one of the driver's devices is added, UMDF calls the **OnDevice** method, passing as parameters pointers to the **IWdfDriver** and **IWdfDeviceInitialize** interfaces, which the framework implements. The **Skeleton** driver does not support physical hardware, so its **OnDeviceAdd** method is minimal:

```
HRESULT
CMyDriver::OnDeviceAdd{
    __in IWDFDriver *FxWdDriver,
    __in IWDFDeviceInitialize *FxDeviceInit
}
/*++
    Routine Description:

The FX invokes this method to install our driver on a device
stack. This method creates a device callback object, then
calls the Fx to create an Fx device object and associate the
new callback object with it.

    Arguments:

    FxWdfDriver - the Fx driver object.
    FxDeviceInit - the initialization information for the
    device.

    Return Value:
    Status
--*/
{
    HRESULT hr;

    PCMyDevice device = NULL;

    //
    // TODO: Here is where to do any per device initialization
    // (reading settings from the registry, for example) that is
    // required before you create the device callback object.
    // You can leave initialization of the device callback
    // object itself to the device event handler.
    //


//
// Create a new instance of our device callback object
//

hr = CMyDevice::CreateInstance(FxWdfDriver, FxDeviceInit,
    &device);

    //
    // TODO: Change any per device settings that the object
    // exposes before you call Configure to complete its
    // initialization.
    //
    //
    // If that succeeded then call the device's construct
    // method. The construct method can create queues or
    // other structures that are required for the device object.

    if (S_OK == hr)
    {
        hr = device->Configure();
    }

    //
    // Release the reference on the device callback object
    // now that it's associated with an fx device object.
    //

    if (NULL != device)
{
        device->Release();
    }

    return hr;
}
```

The **Skeleton** driver's **OnDeviceAdd** method calls the **CreateInstance** method on the **CMyDevice** class to instantiate the device callback object. It passes the pointers to the **IWdfDevice-Initialize** and **IWdfDriver** interfaces so that **CreateInstance** can use these UMDF defined interfaces to create and initialize the device object.

By convention, a **CreateInstance** method in the sample represents a factory for building objects of a particular type.

4.7.4 Device CallBack Object

The device callback object represents the device in the driver. The driver creates an instance of this object when its **IDriverEntry::OnDeviceAdd** method is called. The driver implements the **CreateInstance**, **Initialize**, **Configure**, and **QueryInterface** methods for the device callback object.

The code for the device callback object for the **Skeleton** driver is in the **Device.cpp**. This module includes the header files **Internal.h**, which contains driver-specific internal definitions, and **Device.tmh**, which defines tracing information for Event Tracing for Windows (ETW).

4.7.4.1 CreateInstance Method

A driver's **IDriverEntry::OnDeviceAdd** method calls **IDeviceObject::-CreateInstance** to create an instance of the device callback object. This method simply allocates and initializes an instance of the device callback object as follows:

```
HRESULT
CMyDevice::CreateInstance(
    __in IWDFDriver *FxDriver,
    __in IWDFDeviceInitialize *FxDeviceInit,
    __out PCMyDevice *Device
    )
/*++

    Routine Description:

    This method creates and initializes an instance of the
    skeleton Driver's device callback object.

    Arguments:

    FxDeviceInit - the settings for the device.
    Device - a location to store the referenced pointed to the
             device object.
    Return Value:

    Status
--*/
{
    PCMyDevice device;
    HRESULT hr;

    //
    // Allocate a new instance of the device class.
    //

    device = new CMyDevice();

    if (NULL == device)
   {
        Return E_OUTOFMEMORY;
    }
    //
    // Initialize the instance.
    //

    hr = device->Initialize(FxDriver, FxDeviceInit);

    if (S_OK == hr)
    {
        *Device = device;
    }
    else
    {
        device->Release();
    }

    return hr;
}
```

When UMDF calls the **OnDeviceAdd** method, it passes a pointer to the **IWdfDriver** interface and a pointer to the **IWdfDeviceInitialize** interface. These interfaces provide methods through which the driver can initialize per-device-object settings and create a device callback object. **OnDeviceAdd** passes these pointers to **CreateInstance**, which in turn passes them as parameters to the **Initialize** method to initialize the instance.

4.7.4.2 Initialize Method

The **Initialize** method of the device callback object does exactly what its name implies: It initializes the callback object. It also calls the framework to create the framework's device object.

The **initialize** method receives a handle to the framework's **IWdfDeviceInitialize** interface and stores it in **FxDeviceInit**. It uses this handle to call methods on that interface to initialize certain device characteristics that must be set before the framework's device object is created. Such characteristics include the synchronization (locking) model and the Plug and Play features. They also indicate whether the driver is a filter driver, whether the driver controls device power policy, and whether the framework should forward or fail certain request types. The driver must set these values before creating the framework's device object because they determine which callbacks the framework initializes for the device object. The following code shows how to set the needed values:

```
HRESULT
CMyDevice::Initialize(
__in IWDFDriver       *FxDriver,
__in IWDFDeviceInitialize   *FxDeviceInit
)
/*++

   Routine Description:

   This method initializes the device callback object and
   creates the partner device object.

   The method should perform any device specific configuration
   that:
Could fail (these can't be done in the constructor)
must be done before the partner object is created-or-
can be done after the partner object is created and
isn't influenced, by any device level parameters that the
parent (the driver in this case) might set.

   Arguments:

   FxDeviceInit - the settings for this device.

   Return Value:

   status.

--*/
{
   IWDFDevice    *fxDevice;
   HRESULT       hr;

   //
   // Configure things like the locking model before we
   // create our partner device.
   //

   //
   // TODO: Set her locking mode. The skeleton uses device level
   //               locking, but you can choose "none" as well.
   //

   FxDeviceInit->SetLockingConstraint (WdfDeviceLevel);

   //
   // TODO: If you're writing a filter driver indicate that
   // here.
   //
```

```
    // FxDeviceInit->SetFilter();
    //

    //
    // TODO: Any per-device initialization which must be done
    //              before creating the partner object.
    //
    //
    // Create a new FX device object and assign the new
    // callback object to handle any device level events that
    // occur.
    //

    //
    // QueryIUnknown references the IUnknown interface that it
    // returns (which is the same as referencing the device). We
    // pass that to CreateDevice, which takes its own reference
    // if everything works.
    //

    {
       IUnknown *unknown = this->QueryIUnknown();

       hr = FxDriver->CreateDevice(FxDeviceInit, unknown,
             &fxDevice);

       unknown->Release();
    }

    //
    // If that succeeded, then set our FxDevice member variable.
    //

    if (S_OK == hr)
    {
       m_FxDevice = fxDevice;
       //
       // Drop the reference we got from CreateDevice. Since
       // this object is partnered with the framework object
       // they have the same lifespan. There is no need for an
       // additional reference.
       //
    }

    return hr;
}
```

The **Initialize** method first sets the locking model for the driver by calling the **SetLockingConstraint** method of the **IWdf-DeviceInitialize** interface. The locking model determines whether the framework calls the driver's callback methods concurrently on a per-device-object level or not at all. The **Skeleton** driver sets **WdfDeviceLevel**, which means that the framework synchronizes calls to methods at the device object level or lower. Therefore, the driver does not require code to synchronize access to shared data in such methods.

Device-level locking applies to methods on the **IWdfIoQueue** interface, the **IFileCallbackCleanup** interface, and the **IFileCallback-Close** interface. The **IWdfIoQueue** interface is implemented by the I/O queue object, and the **IFileCallbackCleanup** and **IFileCallbackClose** interfaces are implemented by the device object.

The **Skeleton** does not support physical hardware, so it does not set any Plug and Play characteristics. If it supported an actual Plug and Play device, it might also have to specify whether the device is ejectable, lock-able, and other similar settings.

After the driver has set the device characteristics, it can call UMDF to create the framework's device object. The **IWdfDriver::Create-Device** method takes a pointer to the **IWdfDeviceInitialize** interface that was passed to the driver, a pointer to the driver's device callback object, and a location in which to return the handle to the created frame-work device object. To get a pointer to the device callback object, the driver calls **QueryIUnknown** on the current interface. It then passes this pointer when it calls **CreateDevice**. Calling **QueryIUnknown** adds a reference on the **IUnknown** interface it returns—in this case, the driver's callback object interface. After the **CreateDevice** method returns, the driver releases this reference.

If UMDF successfully creates the framework device object, the driver initializes the variable m_FxDevice to hold the pointer to the returned interface. It then calls the **Release** method to release the reference that the **CreateDevice** method added on the returned interface. The m_FxDevice interface has the same lifetime as the framework's **IWdfDevice** interface, so this reference is not required to ensure that the interface persists for the driver.

4.7.4.3 Configure Method

The **Configure** method handles tasks that are related to configuration after the framework and device callback objects have been created. The

Skeleton driver's **OnDeviceAdd** callback invokes the **Configure** method after **CreateInstance** has successfully returned.

In the **Skeleton** driver, **Configure** is a stub. In a driver that handles I/O requests, this method would create and configure I/O queues and queue callback objects.

4.7.4.4 QueryInterface Method

The **QueryInterface** method returns a pointer to any of the device callback object's interfaces. It takes the **interfaceId** as an input parameter and returns a pointer to the interface.

The **Skeleton** driver does not implement any of the event callback interfaces for the device object because it does not support actual hardware. Therefore, it simply returns the pointer to the **IUnknown** interface of the base class **CUnknown**, as follows:

```
return CUnknown::QueryInterface(InterfaceId, object);
```

In a driver that supports actual hardware, this method should validate the input **interfaceId** and return a pointer to the requested interface.

4.8 Using the Skeleton Driver as a Basis for Development

The **Skeleton** driver is designed for use as a basis for UMDF driver development. By customizing the existing code and adding some of your own code, you can create a driver for your specific device. The following tasks are required:

- Customize the exports.
- Customize the Sources file.
- Customize the **INF** file.
- Customize the **Comsup.cpp** file.
- Add device-specific code to the **Driver.cpp** file.
- Add device-specific code to the **Device.cpp** file.

For most drivers, you can use the following files from the **Skeleton** unchanged:

- **Resource.h**
- **Makefile**

- **ComSup.cpp** and **ComSup.h**, which supply basic support for COM
- **DllSup.cpp**, which supports basic **DLL** functions

4.8.1 Customize the Exports File

The file **Exports.def** lists the library and function names that the **DLL** exports. To customize this file, replace the value in the **LIBRARY** statement with the name of the binary file that contains the **DLL**. For example:

```
LIBRARY            "MyDevice.DLL"
```

Your driver must export the **DLLGetClassObject** function, so you can leave the **EXPORTS** area unchanged.

4.8.2 Customize the Sources File

The Sources file defines environment variables and settings that are required to build the driver. It is input to the generic makefile that is supplied with the samples. To create a makefile to build your own driver, you do not edit the generic makefile. Instead, you edit the Sources file.

To customize the Sources file:

- Change the **TARGETNAME** statement to include the name for your driver. For example:

  ```
  TARGETNAME=MyDevice
  ```

- Change the **SOURCES** statement to include the source files for your driver. For example:

  ```
  SOURCES=\
     MyDevice.rc       \
     dllsup.cpp       \
     comsup.cpp       \
     driver.cpp       \
     device.cpp       \
  ```

- Change the **NTTARGETFILES** statement to include the **INF** files and any other miscellaneous files for your driver. For example:

  ```
  NTTARGETFILES=$(OBJ_PATH)\$(0)\UMDFSkeleton_Root.inf\
        $(OBJ_PATH)\$(0)\UMDFSkeleton_OSR.inf
  ```

4.8.3 Customize the INX File

The **INX** file contains the **INF** that is used to install the driver. To use this file as a basis for your own driver's installation, you must change a variety of settings. The following list outlines the types of required changes:

- Change the **[Manufacturer]** section to include the name of your company and the **[Manufacturer.NT$ARCH$]** section to include the name and location of your driver.
- Change the **[SourceDisksFiles]** section to include the name of your **DLL**.
- If your driver is a filter driver, change the **[DDInstall.Services]** section to install the reflector as the top filter driver in the kernel mode device stack. If your driver is a function driver, the **[DDInstall.Services]** section should install the reflector as the service for the device.
- Change the **[DDInstall.Wdf]** section to install your driver as a service and list it in the **UMDFServiceOrder** directive.
- If your driver performs impersonation, add the **UMDF-Impersonation** directive that specifies the maximum impersonation level for the driver.
- In the **[UMDFServiceInstall]** section, change the name of the binary to the name of your driver binary and specify your driver's class ID in the **DriverCLSID** directive.
- In the **UMDriverCopy** section, specify the name of your **DLL**.
- In the **[Strings]** section, change the strings to specify the name of your company, installation medium, and so forth.

Additional changes might be required depending on the type of device that your driver supports or whether yours is a software-only driver.

4.8.4 Customize the Comsup.cpp File

If you change the name of the driver class to something different from **CMyDriver**, you must change the following line in **CClassFactory::CreateInstance** to reflect the new class name:

```
hr = CMyDriver::CreateInstance(&driver);
```

4.8.5 Add Device-Specific Code to Driver.cpp

In the driver.cpp file, you should add code to the **OnDeviceAdd** method to initialize your device and to change any device-specific settings. For example, if your driver must read settings from the registry before initializing the device, it should do so in **OnDeviceAdd**.

The **CreateInstance**, **AddRef**, **Release**, and **QueryInterface** methods from the **Skeleton** driver should suffice for most drivers.

4.8.6 Add Device-Specific Code to Device.cpp

The file **Device.cpp** is where you must do the most work. The **Skeleton** sample does not support an actual device, so it implements very few of the interfaces and callback objects that are required for the typical device.

In the **Initialize** method, you should set the locking constraint for your driver. The locking constraint determines whether your driver's callback methods can be called concurrently or whether only one such method at a time can be active. Note, however, that the locking model applies strictly to the number of callback methods that are concurrent; it does not limit the number of I/O requests that can be active in your driver at one time.

If your driver is a filter driver, you should indicate that in the **Initialize** method as well, by calling the **SetFilter** method of the **IDeviceInitialize** interface.

In the **Configure** method, you create the I/O queues for the driver. Because the **Skeleton** driver does not handle actual I/O requests, it does not set up any queues. Most drivers, however, implement one or more queues through which UMDF dispatches I/O requests. To create a queue, the driver calls the **IWdfDevice::CreateIoQueue** method and specifies how the queue dispatches requests to the driver: in parallel as soon as they arrive, sequentially (one at a time), or only when the driver calls a method on the queue to request one. The driver then calls **IWdfIoQueue::ConfigureRequestDispatching** to specify the types of requests that should be directed to the queue. The driver must also implement methods in the **IQueueCallbackXxx**, **IRequestCallbackXxx**, and **IFileCallbackXxx** interfaces as appropriate to handle the requests that are directed to its queues.

Finally, a driver that supports a Plug and Play device typically must implement the **IPnPCallback** interface and possibly the

IPnPCallbackHardware and **IPnPCallbackSelfManagedIo** interfaces as well.

You should also either update the header file **Internal.h** or add your own device-specific header file with any additional definitions pertinent to your device-specific code.

CHAPTER 5

Using COM to Develop UMDF Drivers

The Component Object Model (COM) is a specification for a way of building applications based on using components. In the traditional sense, applications were divided into files, modules, or classes. They were then linked to form a monolithic application. A component, however, is like a small application that comes packaged as a binary piece of code that is compiled, linked, and ready to use. This piece of code then links with other components at run-time to form an application. One of the great features of this approach is that one can change or enhance the application by simply replacing one of the components. This chapter introduces COM and its use in creating UMDF drivers. For a more detailed discussion on COM, see the Bibliography for a reference to the book *Inside COM* (Rogerson 1997).

For our work in the UMDF driver development, we must use a number of COM objects that are part of the WDF framework. A number of callback objects are needed to be created based on COM-based callback objects. In the UMDF driver development, we will not use the COM run-time, which contained a good deal of complexity. In the UMDF driver development, we use the essential core of the COM programming model. This keeps the UMDF driver fairly lightweight and thus relatively easy to implement.

5.1 Getting Started

In general, the UMDF drivers are programmed using C++, and the COM objects are developed and also written in C++. It is good to have an understanding of class structure such as the struct and class keywords, public and private members, static methods, constructors, destructors, and pure abstract classes. Also, you should understand object creation, which includes base and derived classes, multiple inheritance, and pure virtual methods.

See the Bibliography for a reference to *C++ How to Program, Seventh Edition* (Deitel 2009) to review or get an understanding of these concepts. In UMDF drivers, the operator overloading or templates are not necessary. The UMDF drivers could use, but are not required to use, the C++ standard template libraries. This includes the Standard Template Library (STL) and the Active Template Library (ATL). We do not show examples of the use of these libraries.

5.1.1 COM Fundamentals

Let's look at some of the fundamental aspects of COM to get us started:

- **IUnknown** is the core COM interface. All COM interfaces derive from this interface. Every COM object exposes this interface and it is essential to the object's operation.
- One of the significant differences between objects in COM and other Object Oriented Programming (OOP) models is that there are no fundamental object pointers in COM. COM exposes interfaces that are groups of related methods. Objects typically expose at least two and sometimes many interfaces. Thus, when you obtain a COM object, you are given a pointer to one of the object's interfaces not the object itself.
- **Globally Unique Identifiers** (GUIDs) are used by COM to uniquely identify COM interfaces. Some COM objects have GUID identifiers, which are referred to as CLSIDs. GUIDs are referred to as IIDs. We can request an interface pointer using these IIDs. COM uses GUIDs for two primary purposes:
  - **Interface ID (IID)**—An IID is a GUID that uniquely identifies a particular COM interface. The interface always has the same IID, regardless of which object exposes it.
  - **Class ID (CLSID)**—A CLSID is a GUID that identifies a particular COM object. CLSIDs are required for COM objects that are created by a class factory, but optional for objects that are created in other ways. With UMDF, only the driver callback object has a class factory or a CLSID.

  To simplify using GUIDs, an associated header file usually defines friendly names that conventionally have a prefix of either **IID_** or **CLSID_** followed by the descriptive name. For example, the friendly name for the GUID that is associated with **IDriverEntry** is **IID_IDriveEntry**. For convenience, the UMFD documentation usually refers to interfaces by the name used in their implementation, such as **IDriverEntry**, rather than the **IID**.

- Any of the methods on an interface can be used with an interface pointer. If you want access to a method on another interface you must obtain another interface pointer. That is done using the **IUnknown::QueryInterface** method.
- There is no public data member's exposure in COM objects. Public data is exposoured through methods we call accessors. In UMDF, we use a Get/Retrieve or Set/Assign prefix for its read and write accessors, respectively. Figure 5.1 shows the logical relationship between an object and its contents.

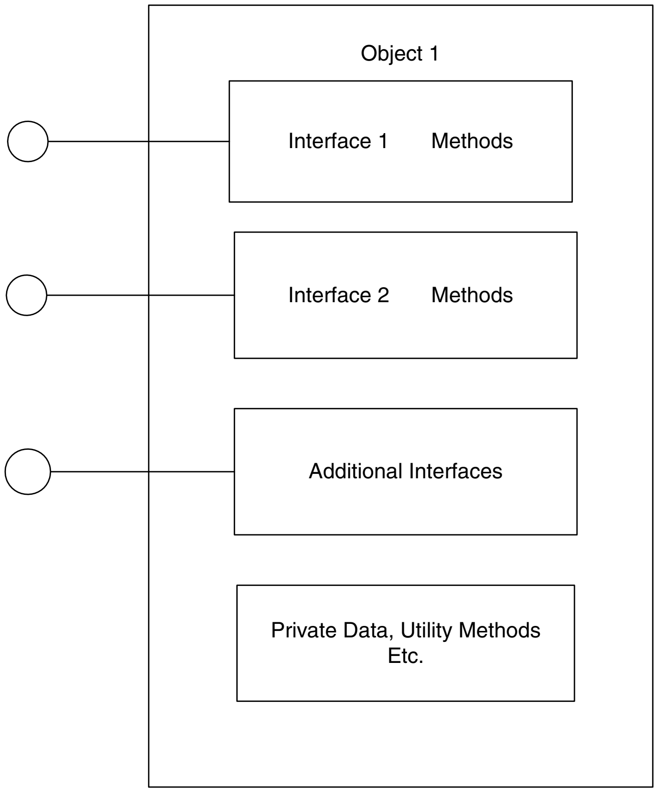

Figure 5.1 COM Objects, Interfaces, and Methods

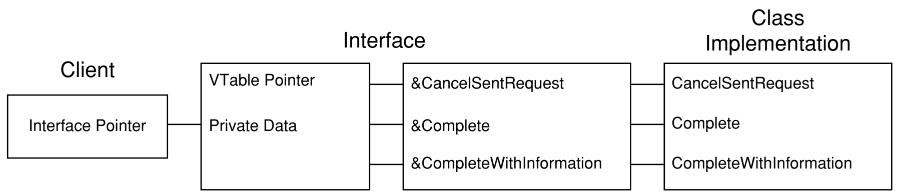

Figure 5.2 VTable and Interface Pointers

All access to COM objects is through a virtual function table—commonly called a VTable—that defines the physical memory structure of the interface. The VTable is an array of pointers to the implementation of each of the methods that the interface exposes. When a client gets a pointer to an interface, it is actually a pointer to the VTable pointer, which in turn points to the method pointer. For example, Figure 5.2 shows the memory structure of the VTable for **IWDFIoRequest.**

The VTable is exactly the memory structure that many C++ compilers create for a pure abstract base class. This is one of the main reasons that COM objects are normally implemented in C++, with interfaces declared as pure abstract base classes. You can then use C++ inheritance to implement the interface in your objects, and the VTable is created for you by the compiler.

5.1.2 HRESULT

Before we look at using COM objects, let's look at the return from a COM method. COM methods often return a 32-bit type called an **HRESULT**. It's similar to the **NTSTATUS** type that Kernel Mode Driver routines use as a return value and is used in much the same way. Figure 5.3 shows the layout of an **HRESULT**.

The **HRESULT** type has three fields:

- Severity, which is essentially a Boolean value that indicates success or failure
- Facility, which can usually be ignored
- Return code, which provides a more detailed description of the results

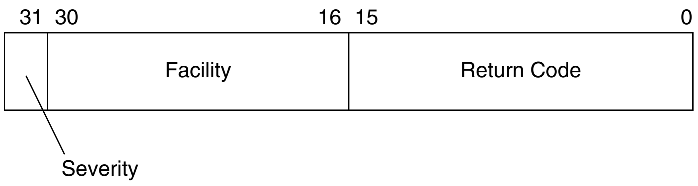

Figure 5.3 HRESULT Layout

As with **NTSTATUS** values, it's rarely necessary to parse the **HRESULT** and examine the individual fields. Standard **HRESULT** values are defined in header files and described on method reference pages. By convention, success codes are assigned names that begin with **S_** and failure codes with **E_**. For example, **S_OK** is the standard **HRESULT** value for simple success.

It's important not to think of **HRESULT** as error values. Methods often have multiple return values for success and for failure. **S_OK** is the usual return value for success, but methods sometimes return other success codes, such as **S_FALSE**.

The severity value is all that is needed to determine whether the method simply succeeded or failed. Rather than parse the **HRESULT** to get the Severity value, COM provides two macros that work much like the **NT_SUCCESS** macro that is used to check **NTSTATUS** values for success or failure. For an **HRESULT** return value of **hr**:

```
FAILED(hr)
   Returns TRUE if the Severity code for hr indicates
   failure and FALSE if it indicates success.

SUCCEEDED(hr)
   Returns FALSE if the Severity code for hr indicates
   failure and TRUE if it indicates success.
```

You can examine the **HRESULT's** return code to determine whether a failure is actionable. Usually, you just compare the returned **HRESULT** to the list of possible return values on the method's reference page. However, be aware that those lists are often incomplete. They typically have only those **HRESULTs** that are specific to the method or standard **HRESULTs** that have some method-specific meaning. The method might also return other **HRESULTs**.

Always test for simple success or failure with the **SUCCEEDED** or **FAILED** macros, whether or not you test for specific **HRESULT** values. Otherwise, for example, if you test for success by comparing the **HRESULT** to **S_OK** and the method unexpectedly returns **S_FALSE**, your code will probably fail.

Although **NTSTATUS** and **HRESULT** are similar, they are not interchangeable. Occasionally, information in the form of an **NTSTATUS** value must be returned as an **HRESULT**. In that case, use the **HRESULT_FROM_NT** macro to convert the **NTSTATUS** value into an equivalent **HRESULT**. However, do not use this macro for an **NTSTATUS** value of **STATUS_SUCCESS**. Instead, return the **S_OK HRESULT** value. If you need to return a Windows error value, you can convert it to an **HRESULT** with the **HRESULT_FROM_WIN32** macro.

5.2 Using UMDF COM Objects

A process that uses a COM object is known as a COM client. Both UMDF drivers and the UMDF run-time function as COM clients. UMDF drivers interact with UMDF run time by using UMDF-provided COM objects. For example, the UMDF device object represents the device, and drivers can use the object for tasks such as setting or retrieving the device's Plug and Play state.

The UMDF run time interacts with drivers through the drive-provided COM-based callback objects. For example, a driver can create one or more queue callback objects to handle I/O requests. The UMDF run time uses those objects to pass request to the driver.

After you get a pointer to an interface, you can call the interface methods by using the same syntax that is used for a pointer to a C++ method. For example, if **pWdfRequest** is a pointer to an **IWDFloRequest** interface, the following code is an example of how to invoke the interface's **Send** method:

```
HRESULT hr;

hr = pWdfRequest- >Send( m_pIUsbTargetDevice,

                                          WDR_REQUEST_SEND_OPTION_SYNCHRONOUS,

                                                0);
```

The method's return value is an **HRESULT**, a typical return type for COM methods. **HRESULT** is similar to the **NTSTATUS** type that Kernel Mode Drivers use as a return value and is used in much the same way. It is important not to think of **HRESULT** as error values. Methods sometimes have multiple return values for success as well as for failure. You can determine the result of calling a method by comparing the returned **HRESULT** to the list of possible values in the reference documentation. However, be aware that these lists are not always complete. Use the error-checking macros that are discussed shortly to ensure that you do not miss a possible return value.

You can also test an **HRESULT** for simple success or failure. COM provides two macros for that purpose that work much like the **NT_SUCCESS** macro. For an **HRESULT** return value of **hr**:

- **FAILED(hr)** returns **TRUE** for failure and **FALSE** for success.
- **SUCCEEDED(hr)** returns **FALSE** for failure and **TRUE** for success.

Although **NTSTATUS** and **HRESULT** are similar, they are not interchangeable. Occasionally, information in the form of an **NTSTATUS** value must be returned as an **HRESULT**. In that case, you can use the **HRESULT_FROM_NT** macro to convert the **NTSTATUS** into an equivalent **HRESULT**. Do not use this macro for an **NTSTATUS** value of **STATUS_SUCCESS**. In that case, return the **S_OK HRESULT** value.

5.2.1 Obtaining an Interface on a UMDF Object

You can obtain an interface on a UMDF object in one of three ways:

- The UMDF run time passes an interface pointer in to one of the driver's callback methods.
- The driver creates a new WDF object by calling a UMDF object creation method.
- The driver calls **IUnknown::QueryInterface** to request a new interface from an existing WDF object.

You can also receive an Interface through a Driver Method. The first case is the simplest. For example, when the UMDF run time calls a driver's **IDriverEntry::OnDeviceAdd** method, it passes a pointer to the device object's **IWDFDriver** interface.

The following example shows this activity:

```
HRESULT CMyDriver::OnDeviceAdd(
            __in IWDFDriver *FxWdfDriver,
            __in IWDFDeviceInitialize *FxDeviceInit
            )
{
   // Install the driver in the device stack
}
```

You can then use **FxWdfDriver** to access the methods on the driver object's **IWDFDriver** interface. Do not release **FxWdfDriver** when you are finished with it. The caller ensures that the object remains valid during the scope of the method call.

Another way to create a WDF object is by calling the appropriate UMDF object creation method. For example, to create a request object, call the UMDF device object's **IWDFDevice::CreateRequest** method. If you look at the UMDF reference in the Windows Driver Kit (WDK), you will find syntax like that for **IWDFDevice::CreateRequest**:

```
HRESULT CreateRequest(
           IN IUnknown*  pCallbackInterface,
           IN IWdfObject*  pParentObject,
           OUT IWDFIoRequest**  ppRequest
           );
```

ppRequest is an **OUT** parameter that provides an address at which the **CreateRequest** method can store a pointer to the newly created request object's **IWDFObject** interface. The following procedure and sample show how to handle such parameters, by using a call to **CreateRequest** by the UMDF's **fx2_driver** sample as an example.

We would declare a variable, **pWdfRequest**, to hold a pointer to **IWDFIoRequest**.

Then we would pass a reference to **pWdfRequest** to **CreateRequest** as follows:

```
   IWDFIoRequest *pWdfRequest = NULL;
   …
   hr = m_FxDevice- >CreateRequest ( NULL, NULL, &pWdfRequest);
```

When **CreateRequest** returns, **pWdfRequest** holds a pointer to an **IWDFIoRequest** interface. When the caller has finished with **pWdf-Request**, it should release the interface pointer by calling **IUnknown::Release**.

Another approach is to call **QueryInterface** to request a new interface. Objects can expose more than one interface. Sometimes, you have a pointer to one interface and need a pointer to another interface on the same object. In that case, call **IUnknown::QueryInterface** to request the desired pointer. Pass **QueryInterface** the IID of the desired interface and the address of the interface pointer, and **QueryInterface** returns the requested pointer. When the caller is finished with the interface pointer, the caller should release it. The following is an example:

```
VOID CMyDevice::StartTarget ( IWDFIoTarget * pTarget)
{
    IWDFIoTargetStateManagement * pStateMgmt = NULL;
    HRESULT hrQI   =
              pTarget->QueryInterface(IID_PPV_ARGS(&pStateMgmt));
    …
}
```

This example requests an **IWDFIoTargetStateManagement** interface pointer from the UMDF's I/O target object. It uses the **IID_PPV_ARGS** macro—declared in objbase.h—which takes an interface pointer and produces the correct arguments for **QueryInterface**.

QueryInterface belongs to the **IUnknown** interface. However, as shown earlier, there is no need to have an explicit pointer to an object's **IUnknown** interface to call **QueryInterface**. All interfaces inherit from **IUnknown**, so you can use any interface to call **QueryInterface**.

5.2.2 Reference Counting

Unlike C++ objects, a client does not directly manage the lifetime of a COM object. Instead, a COM object maintains a reference count on itself. When a client creates a new object with an object-creation method, the object has a reference count of 1. Each time the client requests an additional interface on the object, the object increments the reference count. When a client is finished with an interface, it releases the interface pointer, which decrements the reference count. When all the interface pointers on the object have been released, the reference count is zero and the object destroys itself.

You must be extremely careful about handling reference counts when you use or implement COM objects. Although clients do not explicitly destroy COM objects, there is no garbage collection to take care of the problem automatically as there is with managed code. A common mistake is to fail to release an interface. In that case, the reference count never goes to zero and the object remains in memory indefinitely. Conversely, releasing the interface pointer too many times causes the object to be destroyed prematurely, which can cause a crash. Failure to correctly manage reference counts is a common cause of memory leaks in COM-based applications, along with a variety of other problems. Even worse, bugs that are caused by mismanaged reference counts can be very difficult to locate.

The following are some basic rules for reference counting:

- Release any interface pointer that is passed to you as an **OUT** parameter when you are finished with it by calling **IUnknown::-Release**. Do not release pointers that are passed as **IN** parameters. A common practice to ensure that all interface pointers are properly released is to initialize all pointers to **NULL**. Then set them to **NULL** again when they are released. That convention allows you to test all the interface pointers in your cleanup code; any non-**NULL** pointers are still valid and should be released.
- The reference count is usually incremented for you. The main exception is when you make a copy of an interface pointer. In that case, call **IUnknown::AddRef** to explicitly increment the object's reference count. You must then release the pointer when you are finished.
- When you discover that the driver has reference counting problems, do not attempt to fix them by simply adding calls to **AddRef** or **Release**. Make sure that the driver is acquiring and releasing references according to the rules. Otherwise, you may find, for example, that the **Release** calls that you added to solve a memory leak occasionally deletes the object prematurely and instead causes a crash.

As with **QueryInterface**, you do not need a pointer to the object's **IUnknown** interface to call **AddRef** or **Release**. You can call these methods from any of the object's interfaces.

5.3 Basic Infrastructure Implementation

In this section, we will discuss the required basic infrastructure to support UMDF drivers. A good starting point for your implementation is to take the sample in this book and modify that code to suit your driver's needs. That code should require at most only modest changes to adapt it to your driver's requirements.

5.3.1 DllMain

A dynamic-link library (DLL) can contain any number of in-process COM objects, but it must have a single entry point that is named **DllMain**. Windows calls **DllMain** after the driver binary has been loaded into a host process and before it is unloaded. The function is also called when threads are created or destroyed. The **dwReason** parameter indicates why the function was called.

When a UMDF driver's **DllMain** function is called for DLL loading or unloading, it should perform only simple module-wide initialization and termination tasks, such as initializing or freeing global variables and registering or unregistering Window Software Trace Preprocessor (WPP) tracing. There is a number of things **DllMain** should definitely not do, such as calling **LoadLibrary**.

When a UMDF driver's **DllMain** function is called for thread creation or destruction, it can ignore the call. For more information, see the function's reference page in the Platform Software Developers Kit (SDK). For a typical **DllMain** implementation, see the dllsup.cpp from the UMDF sample code.

5.3.2 DllGetClassObject

Because class factories aren't exported by name, there is no direct way for a client to get access to them. Instead, the DLL exports the **DllGetClassObject** function by name, which allows it to be called by any client with access to the DLL. For many COM DLLs, including the UMDF samples, **DllGetClassObject** is the only function that is listed in the project's .def file to be exported by name from the DLL.

When a client wants to create an instance of one of the COM objects in the DLL, it passes the CLSID of the desired class factory object to **DllGetClassObject** and the IID of the desired interface, usually **IClassFactory**. **DllGetClassObject** creates a new class factory object and returns a pointer to the appropriate interface on the object. The client can then use the **IClassFactory::CreateInstance** method to create an instance of the object. For a typical implementation of

DllGetClassObject, see dllsup.cpp from the UMDF's sample code in the book.

A standard COM server is also required to implement **DllCan UnloadNow** and, optionally, **DllRegisterServer** and **DllUnregister Server**. These exports are not required for UMDF drivers.

5.3.3 Driver Object's Class Factory

Some COM objects must be created by external clients. For UMDF drivers, there is usually only one such object: that is the driver callback object. A COM object that can be created by an external client must have a class factory. This is a small specialized COM object whose sole purpose is to create a new instance of its associated COM object and return a pointer to a specified interface. For a typical implementation of a class factory, see comsup.cpp from the UMDF's sample code.

Class factories usually expose only one interface in addition to **IUnknown**, **IClassFactory**. The **IClassFactory** interface has two members:

- **CreateInstance** creates an instance of the object and returns the requested interface pointer to the client.
- **LockServer** can be used to keep the DLL in memory. UMDF class factories typically have only a token implementation because UMDF does not use **LockServer**.

Some recommendations for implementing **CreateInstance** are as follows.

- Ignore the first parameter. Its purpose is to support COM aggregation, which is not used by UMDF.
- Create a new driver callback object by whatever means is convenient. The sample code puts the object creation code in a static method on the class that implements the callback object.
- Return the appropriate interface as an **OUT** parameter. At this point, the object should have a reference count of 1.

5.3.4 Implementing a UMDF Callback Object

A UMDF driver consists of a collection of COM callback objects. These objects respond to notification by the UMDF run time and allow the driver to process various events, such as read or write requests. All callback objects are in-process COM objects. This means that they are packaged in a DLL and run in the process context of a UMDF host.

The basic requirements for implementing UMDF callback objects are relatively simple and straightforward:

- Implement the **IUnknown** methods to handle reference counting and provide pointers to the object's interfaces.
- Implement the methods of the UMDF callback interfaces that are to be exported by the object.

5.3.4.1 Implementing the UMDF Callback Class

UMDF callback objects are typically implemented as a C++ class that contains the code to support **IUnknown** plus any UMDF interfaces that the object exposes. The UMDF interfaces are declared in wudfdd.h. Following are some of the requirements:

- The class must inherit from every interface that it exposes. However, it can do so indirectly, for example, by inheriting from a class that in turn inherits from one or more interfaces.
- Interfaces are declared as abstract base classes, so the class must implement all the interface methods.
- The class often inherits from a parent class in addition to interfaces. Many of the UMDF samples, for instance, inherit from a parent class, named **CUnknown**, that contains a base implementation of **IUnknown**.
- The class can contain private data members, public methods that are not part of an interface, and so on. These are for internal use and are not visible to clients.
- Constructors are optional. However, if a class has a constructor, it should contain no code in it that might fail. Put any code that can fail in a public initialization method that can be called after object creation.

As we mentioned, a UMDF callback object is typically implemented as a class that inherits from **IUnknown** and one or more object-specific interfaces. Listing 5.1 shows the full declaration of the **CMyDriver** class. The class inherits from a single UMDF interface **IDriverEntry** and inherits from **IUnknown** through the **CUnknown** parent class. For convenience, several of the simpler methods are implemented here, rather than in the associated .cpp file.

Listing 5.1 Declaration of a Driver's Callback Object

```
Class CMyDriver : public Unknown, Public IDriveEntry
{
private:
    IDriverEntry * QueryIDriverEntry (VOID)
    {
        AddRef();
        return static_cast<IDriverEntry*>(this);
    }
    HRESULT initialize(VOID);
public:
    static HRESULT CreateInstance(__out PCMyDriver *Driver);
public:
    virtual HRESULT STDMETHODCALLTYPE OnInitialize(__in
            IWDFDriver *FxWdfDriver)
    {
        UNREFERENCED_PARAMETER (FxWdfDriver);
        return S_OK;
    }
    virtual HRESULT STDMETHODCALLTYPE OnDeviceAdd(
            __in IWDFDriver *FwWdfDriver,
            __in IWDFDeviceInitialize *FxDeviceInit);
    virtual VOID STDMETHODCALLTYPE OnDeinitialize(
            __in IWDFDriver *FxWdfDriver
            )
    {
        UNREFERENCED PARAMETER(FxWdfDriver);
        return;
    }
    virtual  ULONG STDMETHODCALLTYPE AddRef (VOID)
    {
        return __super::AddRef();
    }
    virtual ULONG STDMETHODCALLTYPE Release(VOID)
    {
        return __super::Release();
    }
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(
        __in REFID InterfacedId,
        __deref_out PVOID *Object
        );
};
```

IUnknown is the core COM interface; it is exposed by every COM object and is essential to the object's operation. The approach that is used by the UMDF sample code is to have an **IUnknown** base class, called **CUnknown**, plus an implementation for each exposed interface that inherits from the base class.

5.3.4.2 Implementing AddRef and Release

Reference counting is arguably the key task of **IUnknown**. Normally, a single reference count is maintained for the object as a whole. The following are some recommendations for handling **AddRef** and **Release**:

- Have the interface-specific implementations pass their calls to the base implementation and let it handle incrementing or decrementing the reference count for the object.
- Use **InterlockedIncrement** and **InterlockedDecrement** to modify the reference count. This eliminates the possibility of a race condition.
- After the **Release** method decrements the reference count, check to see whether the count has gone to zero. If so, there are no outstanding interface pointers and you can use **delete** to destroy the object.
- Both **AddRef** and **Release** return the current reference count. Use this for debugging purposes.

5.3.5 Implementing QueryInterface

QueryInterface is the fundamental mechanism by which a COM object provides pointers to its interfaces. It responds to a client's request by returning the specified interface pointer. The following are some recommendations for **QueryInterface**:

- **QueryInterface** must check the incoming IID to see if the request is for a supported interface. **IsEqualID** is a utility function declared in guiddef.h that simplifies comparing IIDs.
- If the object supports the requested interface, **QueryInterface** calls **AddRef** to increment the object's reference count and returns the requested interface pointer. To return the pointer, **QueryInterface** casts a **this** pointer to the requested interface type. This cast is required because of the way in which C++ handles multiple inheritances.

- When a client queries for **IUnknown**, an object must always return the same **IUnknown** pointer regardless of which interface **QueryInterface** is called from.

The basic process of implementing UMDF callback interfaces is similar to **IUnknown**. Most of the implementation details are governed by the requirements of the individual methods.

PART III

Kernel Mode Drivers

This page intentionally left blank

CHAPTER 6

WINDOWS 7 KERNEL MODE DRIVERS OVERVIEW AND OPERATIONS

The Kernel Mode Driver Framework (KMDF) is an infrastructure for developing Kernel Mode Drivers. It provides a C-language device-driver interface (DDI) and can be used to create drivers for Windows 7. In essence, the framework is a skeletal device driver that can be customized for specific devices. KMDF implements code to handle common driver requirements. Drivers customize the framework by setting object properties, registering callbacks to be notified of important events, and including code only for features that are unique to their device.

KMDF provides a well-defined object model and controls the lifetime of objects and memory allocations. Objects are organized hierarchically in a parent/child model, and important driver data structures are maintained by KMDF instead of by the driver.

This chapter provides an introduction to the architecture and features of KMDF and to the requirements for drivers that use KMDF (sometimes called KMDF-based drivers or simply DMDF drivers).

As we covered in Part II, the Windows Driver Foundation (WDF) also includes a User Mode Driver Framework (UMDF). If your device does not handle interrupts, perform direct memory access (DMA), or require other kernel mode resources such as nonpaged pool memory, you should consider writing a User Mode Driver instead.

6.1 KMDF Supported Devices

KMDF was designed to replace the Windows Driver Model (WDM). The initial KMDF release supports most of the same devices and device classes as WDM, except in those that are currently supported by miniport models. Table 6.1 lists the device and driver types that KMDF supports.

Table 6.1 Devices and Driver Types That KMDF Supports

Device or Driver Type	Existing Driver Model	Comments
Control and non-Plug and Play drivers	Legacy	Supported
IEEE 1394 client drivers	Depends on device class	Supported for devices that do not conform to existing device class specifications
ISA, PCI, PCMCIA, and secure digital (SD) devices	WDM driver	Supported, if device class or port drivers do not provide the driver dispatch functions
NDIS protocol drivers	WDM upper edge and NDIS lower edge	Supported
NDIS WDM drivers	NDIS upper edge and WDM lower edge	Supported
SoftModem drivers	WDM driver with upper Edge support for TAPI Interface	Supported
Storage class drivers and filter drivers	WDM driver	Supported
Transport driver interface (TDI) client drivers	Generic WDM driver	Supported
USB client drivers	Depends on device class	Supported
Winsock client drivers	WDM driver with a callback interface for device-specific requests	Supported

In general, KMDF supports drivers that conform to WDM, supply entry points for the major I/O dispatch routines, and handle I/O request packets (IRPs). For some device types, device class and port drivers supply driver dispatch functions and call back to a miniport driver to handle specific I/O details. Such miniport drivers are essentially callback libraries and are not currently supported by KMDF. In addition, KMDF does not support device types that use the Windows imaging architecture (WIA).

6.2 KMDF Components

KMDF is distributed as part of the Windows Driver Kit (WDK) and consists of header files, libraries, sample drivers, development tools, public debugging symbols, and tracing format files. By default, KMDF is installed in the WDF subdirectory of the WDK root installation directory. KMDF-based drivers are built in the WDK build environment. Table 6.2 lists the KMDF components that are installed as part of WDF.

Table 6.2 KMDF Component Names

Component	Location	Description
Header files	wdf/inc	Header files required to build KMDF drivers
Libraries	wdf/lib	Libraries for x86, x64, and Intel Itanium architectures
Sample drivers	wdf/src	Sample drivers for numerous device types; most are ported from Windows Device Kit (DDK) WDM samples
Tools	wdf/bin	Tools for testing, debugging, and installing drivers, includes the redistributable KMDF co-installer, WdfConinstallernn.dll
Debugging symbols	wdf/symbols	Public symbol database (.pdb) files for KMDF libraries and co-installer for checked and free builds
Tracing format files	wdf/tracing	Trace format files for the trace messages generated by KMDF libraries and co-installer

To aid in debugging, KMDF is distributed with free and checked builds of the run-time libraries and loader, along with corresponding symbols. However, Microsoft does not provide a checked version of the redistributable co-installer itself.

6.3 KMDF Driver Structure

A KMDF driver consists of a **DriverEntry** function that identifies the driver as based on KMDF, a set of callback functions that KMF calls so that the driver can respond to events that affect its device, and other driver-specific utility functions. Nearly every KMDF driver must have the following:

- A **DriverEntry** function, which represents the driver's primary entry point
- An **EvtDriverDeviceAdd** callback, which is called when the Plug and Play manager enumerates one of the driver's devices (not required for drivers that support non-Plug and Play devices)
- One or more **EvtIo*** callbacks, which handle specific types of I/O requests from a particular queue

Drivers typically create one or more queues into which KMDF places I/O requests for the driver's device. A driver can configure the queues by type of request and type of dispatching.

A minimal Kernel Mode Driver for a simple device might have these functions and nothing more. KMDF includes code to support default power management and Plug and Play operations, so drivers that do not manipulate physical hardware can omit most Plug and Play and power management code. If a driver can use the default, it does not require code for many common tasks, such as passing a power IRP down the device stack. The more device-specific features a device supports and the more functionality the driver provides the more code the driver requires.

6.4 Comparing KMDF and WDM Drivers

The KMDF model results in drivers that are much simpler and easier to debug than WDM drivers. KMDF drivers require minimal common code for default operations because most such code resides in the framework, where it has been thoroughly tested and can be globally updated.

Because KMDF events are clearly and narrowly defined, KMDF-based drivers typically require little code complexity. Each driver callback routine is designed to perform a specific task. Therefore, compared to WDM drivers, KMDF-based drivers have fewer lines of code and virtually no state variables or locks.

As part of the WDF development effort, Microsoft has converted many of the sample drivers that shipped with the Windows DDK from WDM drivers to KMDF drivers. Without exception, the KMDF drivers are smaller and less complex.

Table 6.3 shows "before-and-after" statistics for the PCIDRV, Serial, and OSRUSBFX2 drivers.

Table 6.3 WDM-KMDF Statistics for Sample Drivers

Statistic	PCIDRV		Serial		OSRUSBFX2	
	WDM	KMDF	WDM	KMDF	WDM	KMDF
Total lines of code	13,147	7,271	24,000	17,000	16,350	2,300
Lines of code required for Plug and Play and power management	7,991	1,795	5,000	2,500	8,700	742
Locks and synchronization	8	3	10	0	9	0
State variables required for Plug and Play and power management	30	0	53	0	21	0

The PCIDRV sample supports the Intel E100B NIC card. The WDM and KMDF versions are functionally equivalent. The Serial sample supports a serial device. In this case, the WDM sample supports a multiport device, but the KMDF sample supports only a single port. However, the statistics for the WDM driver do not include code, locks, or variables that are required solely to support multiport devices, so the statistics are comparable. The OSRUSBFX2 sample supports the USB-FX2 board built by OSR. The WDM and KMDF versions are functionally equivalent.

As the table shows, converting these drivers from WDM to KMDF resulted in significant reductions in the lines of code, particularly for Plug and Play and power management. The KMDF samples also require fewer locks and synchronization primitives and state variables. The following information shows more detail with respect to these requirements:

- **Lines of code—**The KMDF drivers require significantly fewer lines of code both overall and to implement Plug and Play and power management. Less code means a less complex driver with fewer opportunities for error and a smaller executable image.
- **Locks and synchronization primitives—**Not only are the KMDF drivers smaller, but in all three cases the number of locks and synchronization primitives has been reduced significantly. This change is important because it eliminates a common source of driver problems. WDM drivers use locks to synchronize I/O queues with Plug and Play and power operations and often supply locks to manage I/O cancellation. The locking scenarios typically involve one or more race conditions and can be difficult to implement correctly. KMDF drivers can be implemented with few such locks because the framework provides the locking.
- **State variables—**The number of state variables that are required for Plug and Play and power management is a measure of the complexity of the Plug and Play and power management implementation within the driver. A WDM driver receives Plug and Play and power management requests from the operating system in the form of IRPs. When such a driver receives a Plug and Play or power IRP, it must determine the current state of its device and the system and, based on those two states, must determine what to do to satisfy the IRP. Drivers must handle some IRPs immediately upon arrival as they travel down the device stack, but must handle others only after they have been acted upon by drivers lower in the stack. Consequently, a WDM driver must keep track of numerous details about the current state of its device and of current Plug and Play and power management requests. Tracking this information requires 30 variables in the WDM PCIDRV sample, 53 in the Serial sample, and 21 in the OSRUSBFX2 sample.

The KMDF versions of the three sample drivers require no state variables. The KMDF drivers do not maintain such information because the framework does so on their behalf. The framework implements an extensive state machine that integrates Plug and Play and power management operations with I/O operations. A driver provides callbacks that are invoked only when its device requires manipulation. For example, a driver for a device that supports a wake-up signal can register a callback that arms the signal, and KMDF invokes the callback at the appropriate time. By contrast, a WDM driver must determine which power management IRPs require it to arm the signal and at which point in handling those IRPs it should do so.

6.5 Device Objects and Driver Roles

Every driver creates one or more device objects, which represent the driver's roles in handling I/O requests and managing its device. KMDF supports the development of the following types of device objects:

- Filter device objects (filter DOs) represent the role of a filter driver. Filter DOs "filter," or modify, one or more types of I/O requests that are targeted at the device. Filter DOs are attached to the Plug and Play device stack.
- Functional device objects (FDOs) represent the role of a function driver, which is the primary driver for a device. FDOs are attached to the Plug and Play device stack.
- Physical device objects (PDOs) represent the role of the bus driver, which enumerates child devices. PDOs are attached to the Plug and Play device stack.
- Control device objects represent a legacy non-Plug and Play device or a control interface. They are not part of the Plug and Play device stack.

Depending on the design of the device and the other drivers in the device stack, a driver might assume one or more of these roles. Each Plug and Play device has one function driver and one bus driver, but can have any number of filter drivers. In the Plug and Play device stack, a driver sometimes acts as the function driver for one device and as the bus driver for the devices that its device enumerates. For example, a USB hub driver acts as the function driver for the hub itself and the bus driver for each USB device that is attached to the hub. Thus, it creates an FDO for the hub and a PDO for each attached USB device.

6.5.1 Filter Drivers and Filter Device Objects

A filter driver receives one or more types of I/O requests that are targeted at its device, takes some action based on the request, and then passes the request to the next driver in the stack. Filter drivers do not typically perform device I/O themselves; instead, they modify or record a request that another driver satisfies. Device-specific data encryption/decryption software is commonly implemented as a filter driver.

A filter driver adds a filter DO to the Plug and Play device stack. A KMDF driver notifies the framework that it is a filter driver when its device is added to the system, so that KMDF creates a filter DO and sets the appropriate defaults.

Most filter drivers are not "interested" in every request that is targeted at their devices; a filter driver might filter only read requests or only create requests. To simplify filter driver implementation, KMDF dispatches only the types of requests that the filter driver specifies and passes all other requests down the device stack. The filter driver never receives them and so does not require code to inspect them or pass them to another driver.

The sample Firefly, Kbfiltr, and Toaster Filter drivers create filter DOs.

KMDF does not support the development of bus filter drivers. Such drivers are layered immediately above a bus driver that creates a PDO and add their device objects to the stack when Plug and Play manager queries the bus driver for bus relations.

6.5.2 Function Drivers and Functional Device Objects

Function drivers are the primary drivers for their devices. A function driver communicates with its device to perform I/O and typically manages power policy for its device. In the Plug and Play device stack, a function driver exposes an FDO.

To support function drivers, KMDF includes an FDO interface, which defines a set of methods, events, and properties that apply to FDOs during initialization and operation. By using the FDO interface, a driver can

- Register event callbacks that are related to resource allocation for its device.
- Retrieve properties of its physical device.
- Open a registry key.
- Manage a list of child devices, if the device enumerates one or more children.

When the driver creates its device object, KMDF creates an FDO unless the driver notifies it otherwise.

By default, KMDF assumes that the function driver is the power policy manager for its device. If the device supports wake-up signals, the function driver typically also sets power policy event callbacks to implement this feature. All the sample drivers, except the KbFiltr and Firefly drivers, create an FDO.

6.5.3 Bus Drivers and Physical Device Objects

A bus driver typically operates as the function driver for a parent device that enumerates one or more child devices. The parent device might be a bus but could also be a multifunction device that enumerates children whose functions require different types of drivers. In the Plug and Play device stack, a bus driver exposes a PDO.

KMDF defines methods, events, and properties that are specific to PDOs, just as it does for FDOs. By using the PDO interface, a driver can

- Register event callbacks so that the driver can report the hardware resources that its children require.
- Register event callbacks that are related to device locking and ejection.
- Register event callbacks that perform bus-level operations so that its child devices can trigger a wake signal.
- Assign Plug and Play, compatible, and instance IDs to its child devices.
- Set removal and ejection relations for its child devices.
- Notify the system that a child device has been ejected or surprise removed.
- Retrieve and update the bus address of a child device.
- Indicate that the driver controls a raw device. (A raw device is driven directly by a bus driver, without a function driver.)

To indicate that it is a bus driver, a KMDF driver calls one or more of the PDO initialization methods before creating its device object. If the driver indicates that it is driving a raw device, KMDF assumes that the driver is the power policy manager for the device.

Writing bus drivers is much simpler with KMDF than with WDM. KMDF manages the state of the PDO on behalf of the driver, so that the driver is only required to notify KMDF when the device is added or removed. KMDF supports both static and dynamic models for enumerating child devices. If the status of child devices rarely changes, the bus driver should use the static model. The dynamic model supports drivers for devices such as IEEE 1394 buses, where the status of child devices might change at any time.

For bus drivers, KMDF handles most of the details of enumeration, including:

- Reporting children to WDM.
- Coordinating scanning for children.
- Maintaining the list of children.

In addition, the KMDF interface through which drivers report resource requirements is easier to use than that provided by WDM.

The sample KbFiltr, OsrUsbFx2/EnumSwitches, and Toaster bus drivers create PDOs and use both the static and dynamic methods to enumerate their child devices.

6.5.4 Legacy Device Drivers and Control Device Objects

In addition to Plug and Play function, bus, and filter drivers, KMDF supports the development of drivers for legacy devices, which are not controlled by a Plug and Play lifetime model. Such drivers create **control device objects**, which are not part of the Plug and Play device stack.

Plug and Play drivers can also use **control device objects** to implement control interfaces that operate independently of the device stack. An application can send requests directly to the **control device object**, thus bypassing any filtering performed by other drivers in the stack. Such a **control device object** typically has a queue and might sometimes forward requests from that queue to a Plug and Play device object.

Because **control device objects** are not part of the Plug and Play device stack, the driver must notify KMDF when their initialization is complete. In addition, the driver itself must delete the device object when the device has been removed because only the driver knows how to control the lifetime of such a device. The sample NdisProt, NonPnP, and Toaster Filter drivers create **control device objects**.

6.6 KMDF Object Model

KMDF defines an object-based programming model in which object types represent common driver constructs. Each object exports methods (functions) and properties (data) that drivers can access and is associated with object-specific events, which drivers can support by providing event callbacks. The objects themselves are opaque to the driver.

KMDF creates some objects on behalf of the driver, and the driver creates others depending on its specific requirements. The driver also provides callbacks for the events for which the KMDF defaults do not suit its device and calls methods on the object to get and set properties and perform any additional actions. Consequently, a KMDF driver is essentially a **DriverEntry** routine, a set of callback functions that perform device-specific tasks, and whatever utility functions the driver implementation requires.

Framework-base drivers never directly access instances of framework objects. Instead, they reference object instances by handles, which the driver passes as parameters to object methods and KMDF passes as parameters to event callbacks. Framework objects are unique to the framework. They are not managed by the Windows object manager and cannot be manipulated by using the system's **ObXxx** function. Only the framework (and its drivers) can create and operate on them.

6.6.1 Methods, Properties, and Events

Methods are functions that perform an action on an object, such as creating or deleting the object. KMDF methods are named according to the following pattern:

```
WdfObjectOperation
```

Object specifies the KMDF object on which the method operates, and **Operation** indicates what the method does. For example, the **WdfDeviceCreate** method creates a framework device object.

Properties are functions that read and write data fields in an object, thus defining object behavior and defaults. Properties are named according to the following pattern:

```
WdfObject{Set|Get}Data
WdfObject{Assign|Retrieve}Data
```

Object specifies the KMDF object on which the function operates, and **Data** specifies the field that the function reads or writes. Some properties can be read and written without failure, but others can sometimes fail. Functions with **Set** and **Get** in their names read and write fields without failure. The **Set** functions return **VOID**, and the **Get** functions typically return the value of the field. Functions with **Assign** and **Retrieve** in their names read and write fields but can fail. These functions return an **NTSTATUS** value.

For example, the **WDFINTERRUPT** object represents the interrupt object for a device. Each interrupt object is described by a set of characteristics that indicate the type of interrupt (message signaled or IRQ based) and provide additional information about the interrupt. The **WdfInterruptGetInfo** method returns this information. A corresponding method to set the value is not available because the driver initializes this information when it creates the interrupt object and cannot change it during device operation.

Events represent run-time states to which a driver can respond or during which a driver can participate. A driver registers callbacks only for the events that are important to its operation. When the event occurs, the framework invokes the callback, passing as a parameter a handle to the object for which the callback is registered. For example, the ejection of a device is a Plug and Play event. If a device can be ejected, its driver registers an **EvtDeviceEject** callback routine, which performs device-specific operations upon ejection. KMDF calls this routine with a handle to the device object when the Plug and Play manager sends an **IRP_MN_EJECT** request for the device. If the device cannot be ejected, the driver does not require such a callback.

For most events, a driver can either provide a callback routine or allow KMDF to perform a default action in response. For a few events, however, a driver-specific callback is required. For example, adding a device is an event for which every Plug and Play driver must include a callback. The driver's **EvtDriverDeviceAdd** callback creates the device object and sets device attributes.

KMDF events are not related to the kernel-dispatcher events that Windows provides as synchronization mechanisms. A driver cannot create, manipulate, or wait on a KMDF event. Instead, the driver registers a callback for the event and KMDF calls the driver when the event occurs. (For time-related waits, KMDF provides timer objects.)

6.6.2 Object Hierarchy

KMDF objects are organized hierarchically. **WDFDRIVER** is the root object; all other objects are considered its children. For most object types, a driver can specify the parent when it creates the object. If the driver does not specify a parent at object creation, the framework sets the default parent to the **WDFDRIVER** object. Figure 6.1 shows the default KMDF object hierarchy.

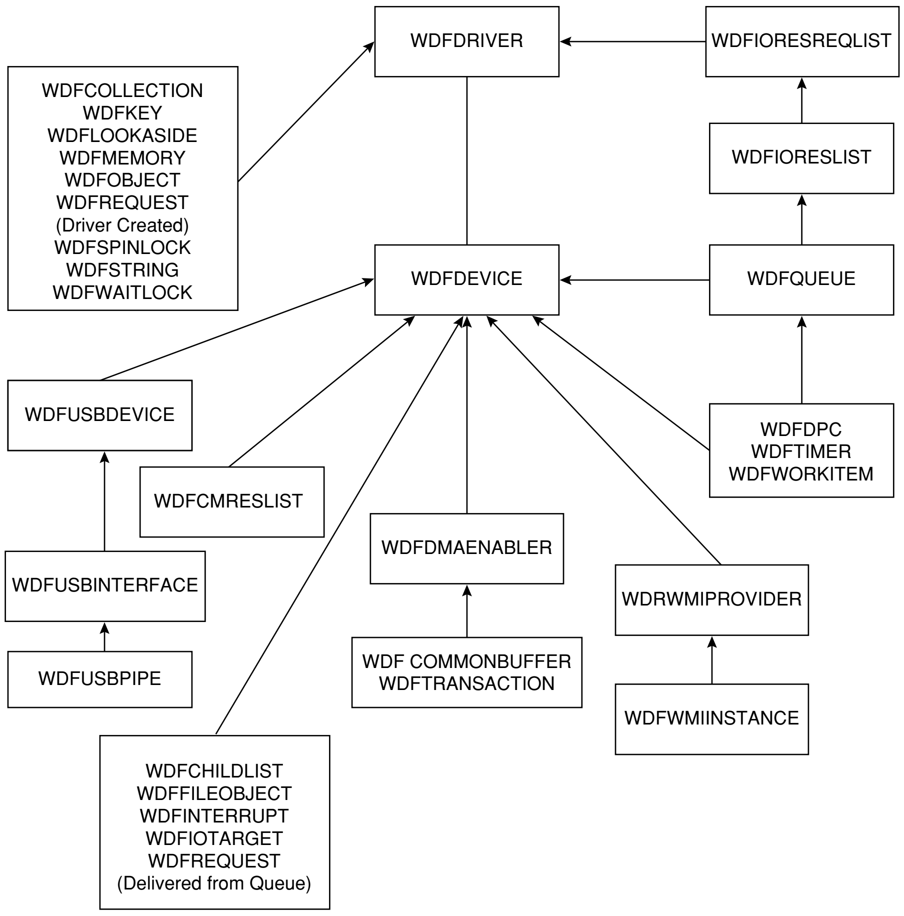
WDFDRIVER
WDFIORESREQLIST
WDFCOLLECTION WDFKEY WDFLOOKASIDE WDFMEMORY WDFOBJECT WDFREQUEST (Driver Created) WDFSPINLOCK WDFSTRING WDFWAITLOCK
WDFIORESLIST
WDFDEVICE
WDFQUEUE
WDFUSBDEVICE
WDFDPC WDFTIMER WDFWORKITEM
WDFCMRESLIST
WDFDMAENABLER
WDFUSBINTERFACE
WDRWMIPROVIDER
WDFUSBPIPE
WDF COMMONBUFFER WDFTRANSACTION
WDFWMIINSTANCE
WDFCHILDLIST WDFFILEOBJECT WDFINTERRUPT WDFIOTARGET WDFREQUEST (Delivered from Queue)

Figure 6.1 Parent-Child Relationships Among the KMDF Objects

For each object, the figure shows which other object(s) must be in its parent chain. These objects are not necessarily the immediate parent but could be the grandparent, great-grandparent, and so forth. For example, Figure 6.1 shows the **WDFDEVICE** object as parent of the **WDFQUEUE** object. However, a **WDFQUEUE** object could be the child of a **WDFIOTARGET** object, which in turn is the child of a **WDFDEVICE** object. Thus, the **WDFDEVICE** object is in the parent chain for the **WDFQUEUE** object.

The object hierarchy affects the object's lifetime. The parent holds a reference count for each child object. When the parent object is deleted, the child objects are deleted and their callbacks are invoked in a defined order. Table 6.4 lists all the KMDF object types.

Table 6.4 KMDF Object Types

Object	Type	Description
Child list	WDFCHILDLIST	Represents a list of the child devices for a device.
Collection	WDFCOLLECTION	Describes a list of similar objects, such as resources or the devices for which a filter driver filters requests.
Device	WDFDEVICE	Represents an instance of a device. A driver typically has one WDFDEVICE object for each device that it controls.
DMA common buffer	WDFCOMMON BUFFER	Represents a buffer that can be accessed by both the device and the driver to perform DMA.
DMA enabler	WDFDMAENABLER	Enables a driver to use DMA. A driver that handles device I/O operations has one WDFDMAENABLER object for each DMA channel within the device.
DMA transaction	WDFDMATRANS-ACTION	Represents a single DMA transaction.
Deferred Procedure Call (DPC)	WDFDPC	Represents a Deferred Procedure Call.
Driver	WDFDRIVER	Represents the driver itself and maintains information about the driver, such as its entry points. Every driver has one WDFDRIVER object.

Object	Type	Description
File	WDFFILEOBJECT	Represents a file object through which external drivers or applications can access the device.
Generic object	WDFOBJECT	Represents a generic object for use as the driver requires.
I/O queue	WDFQUEUE	Represents an I/O queue. A driver can have any number of WDFIOQUEUE objects.
I/O request	WDFREQUEST	Represents a request for device I/O.
I/O target	WDFIOTARGET	Represents a device stack to which the driver is forwarding an I/O request.
Interrupt	WDFINTERRUPT	Represents a device's interrupt object. Any driver that handles device interrupts has one WDFINTERRUPT object for each IRQ or message-signaled interrupt (MSI) that the device can trigger.
Look-aside list	WDFLOOKASIDE	Represents a dynamically sized list of identical buffers that are allocated from the paged or nonpaged pool. Both the WDFLOOKASIDE object and its component memory buffers can have attributes.
Memory	WDFMEMORY	Represents memory that the driver uses, typically an input or output buffer that is associated with an I/O request.
Registry key	WDFKEY	Represents a registry key.
Resource list	WDFCMRESLIST	Represents the list of resources that have actually been assigned to the device.
Resource range list	WDFIORESLIST	Represents a possible configuration for a device.
Resource requirements list	WDFIORESREQLIST	Represents a set of I/O resource lists, which comprises all possible configurations for the device. Each element of the list is a WDFIORESLIST object.
String	WDFSTRING	Represents a counted Unicode string.
Synchro-nization: spin lock	WDFSPINLOCK	Represents a spin lock, which synchronizes access to data DISPATCH_LEVEL.

(continues)

Table 6.4 KMDF Object Types *(continued)*

Object	Type	Description
Synchronization: wait lock	WDFWAITLOCK	Represents a wait lock, which synchronizes access to data at PASSIVE_LEVEL.
Timer	WDFTIMER	Represents a timer that fires either once or periodically and causes a callback routine run.
USB device	WDFUSBDEVICE	Represents a USB device.
USB interface	WDFUSBINTERFACE	Represents an interface on a USB device.
USB pipe	WDFUSBPIPE	Represents a pipe in a USB interface.
Windows Management Instrumentation (WMI) instance	WDFWMIINSTANCE	Represents an individual WMI data block that is associated with a particular provider
WMI provider	WDFWMIPROVIDER	Represents the schema for WMI data blocks that the driver provides.
Work item	WDFWORKITEM	Represents a work item, which runs in a system thread at PASSIVE_LEVEL.

6.6.3 Object Attributes

Every KMDF object is associated with a set of attributes. The attributes define information that KMDF requires for objects, as listed in Table 6.5.

Table 6.5 KMDF Object Attributes

Field	Description
ContextSizeOverride	Size of the context area; overrides the value in **ContextTypeInfo->ContextSize.**
ContextTypeInfo	Pointer to the type information for the object context area.
EvtCleanupCallback	Pointer to a callback routine that is invoked to clean up the object before it is deleted; the object might still have references.

Field	Description
EvtDestroyCallback	Pointer to a callback routine that is invoked when the reference count reaches zero for an object that is marked for deletion.
ExecutionLevel	Maximum interrupt request level (IRQL) at which KMDF can invoke certain object callbacks.
ParentObject	Handle to the object's parent.
Size	Size of the object
SynchronizationScope	Level at which certain callbacks for this object are synchronized; applies only to driver, device, and file objects.

The framework supplies defaults for most attributes. A driver can override these defaults when it creates the object by using the **WDF_OBJECT_ATTRIBUTES_INIT** function.

6.6.4 Object Context

Every instance of a KMDF object can have one or more object context areas. This area is a driver-defined storage area for data that is related to a specific instance of an object, such as a driver-allocated lock or event for the object. The size and layout of the object context area are determined by the driver. When the driver creates the object, it initializes the context area and specifies its size and type. The driver can create additional context areas after the object has been created. For a KMDF device object, the object context area is the equivalent of the WDM device extension.

When KMDF creates the object, it allocates memory for context areas from the nonpaged pool and initializes them according to the driver's specifications. When KMDF deletes the object, it deletes the context areas, too. The framework provides macros to associate a type and a name with the context area and to create a named accessor function that returns a pointer to the context area.

If you are familiar with WDM, this design might seem unnecessarily complicated. However, it provides flexibility in attaching information to I/O requests as they flow through the driver. In addition, it enables different libraries to have their own separate context for an object. For example, an IEEE 1394 library could track a WDFDEVICE object at the same time that the device's function driver tracks it, but with separate contexts. Within a driver, the context area enables a design pattern that is similar to inheritance. If the driver uses a request for several different tasks, the request object can have a separate context area to each task. Functions that are related to a specific task can access their own contexts and do not require any information about the existence or contents of any other contexts.

6.6.5 Object Creation and Deletion

To create an object, KMDF does the following:

- Allocates memory from the nonpaged pool for the object and its context areas.
- Initializes the object's attributes with default values and the driver's specifications (if any).
- Zeroes the object's context areas.
- Configures the object by storing pointers to its event callbacks and setting other object-specific characteristics.

If object initialization fails, KMDF deletes the object and any children that have already been created.

To initialize object attributes and configuration structures, a driver invokes KMDF initialization functions before it calls the object-creation methods. KMDF uses the initialized attributes and structures when it creates the object.

KMDF maintains a reference count for each object and ensures that the object persists until all references to it have been released. If the driver explicitly deletes an object (by calling a deletion method), KMDF marks the object for deletion but does not physically delete it until its reference count reaches zero.

Drivers do not typically take out references on the objects that they create, but in some cases (such as when escaping directly to WDM) such references are necessary to ensure that the object's handle remains valid. For example, a driver that sends asynchronous I/O requests might take out a reference on the request object to guard against race conditions during cancellation. Before the request object can be deleted, the driver must release this reference.

Object deletion starts from the object farthest from the parent and works up the object hierarchy toward the root. KMDF takes the following steps to delete an object:

- Starting with the child object farthest from the parent, calls the object's **EvtCleanupCallback**. In this routine, drivers should perform any cleanup tasks that must be done before the object's parent is deleted. Such tasks might include releasing explicit references on the object or a parent object. Note that when the **EvtCleanupCallback** function runs, the object's children still exist; even though their **EvtCleanupCallback** functions have already been invoked.
- When the object's reference count reaches zero, calls the object's **EvtDestroyCallback**, if the driver has registered one.
- Deallocates the memory that was allocated to the object and its context area.

KMDF always calls the **EvtCleanupCallback** routines of child objects before calling those of their parent objects, so drivers are guaranteed that the parent object still exists when a child's **EvtCleanupCallback** runs. This guarantee does not apply to **EvtDestroyCallbacks**, however; KMDF can call the **EvtDestroyCallback** routines in any order, so that the **EvtDestroyCallback** for a parent might be called before that of one of its children.

Drivers can change the parent of most KMDF objects by setting the **ParentObject** attribute. By setting the parent/child relationships appropriately, a driver can avoid taking out explicit references on related objects and can instead use the hierarchy and the associated callbacks to manage the object's lifetime.

6.7 KMDF I/O Model

KMDF established its own dispatch routines that intercept all IRPs that are sent to the driver. Figure 6.2 shows the overall flow of I/O through the KMDF library and driver.

When an IRP arrives, KMDF directs it to one of the following components for processing:

- I/O request handler, which handles requests that involve device I/O.
- Plug and Play/power request handler, which handles Plug and Play and power request (**IRP_MJ_PNP** and **IRP_MJ_POWER** requests) and notifies other components of changes in device status.

- WMI handler, which handles WMI and event-tracing request (**IRP_MJ_SYSTEM_CONTROL** requests).

Each component takes one or more of the following actions for each request:

- Raises one or more events to the driver.
- Forwards the request to another internal handler or I/O target for further processing.
- Completes the request based on its own action.
- Completes the request as a result of a driver call.

If the request has not been processed when it reaches the end of frameworks processing, KMDF takes an action that is appropriate for the type of driver. For function and bus drivers, KMDF completes the request with the status **STATUS_INVALID_DEVICE_REQUEST**. For filter drivers, KMDF automatically forwards the request to the default I/O

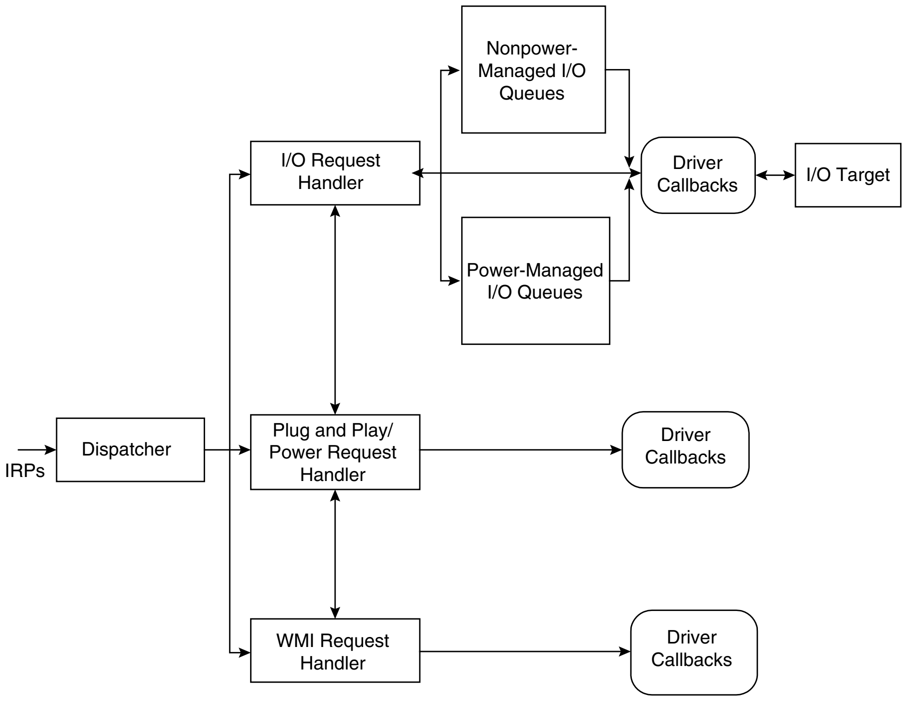

Figure 6.2 KMDF I/O Flow

target (the next lower driver). The next sections describe how each of the three components processes I/O requests.

6.7.1 I/O Request Handler

The I/O request handler dispatches I/O requests to the driver, manages I/O cancellation and completion, and works with the Plug and Play/power handler to ensure that the device state is compatible with performing device I/O.

Depending on the type of I/O request, the I/O request handler either queues the request or invokes an event callback that the driver registered for the request.

6.7.1.1 Create, Cleanup, and Close Requests

To handle create events, a driver can either configure a queue to receive the events or can supply an event callback that is invoked immediately. The driver's options are the following:

- To be called immediately, the driver supplies an **EvtDeviceFileCreate** callback and registers it from the **EvtDriverDeviceAdd** callback by calling **WdfDeviceInitSetFileObjectConfig**.
- To configure a queue to receive the requests, the driver calls **WdfDeviceConfigureRequestDispatching** and specifies **WdfRequestTypeCreate**. If the queue is not manual, the driver must register an **EvtIoDefault** callback, which is called when a create request arrives.

Queuing takes precedence over the **EvtDeviceFileCreate** callback—that is, if the driver both registers for **EvtDeviceFileCreate** events and configures a queue to receive such requests, KMDF queues the requests and does not invoke the callback. KMDF does not queue create requests to a default queue; the driver must explicitly configure a queue to receive them.

In a bus or function driver, if a create request arrives for which the driver has neither registered an **EvtDeviceFileCreate** callback function nor configured a queue to receive create requests, KMDF opens a file object to represent the device and completes the request with **STATUS_SUCCESS**. Therefore, any bus or function driver that does not accept create or open requests from user mode application—and thus does not register a device interface—must register an **EvtDeviceFileCreate** callback that explicitly fails such requests. Supplying a callback to fail create requests ensures that a rogue user mode application cannot gain access to the device.

If a filter driver does not handle create requests, KMDF by default forwards all create, cleanup, and close requests to the default I/O target (the next lower driver). Filter drivers that handle create requests should perform whatever filtering tasks are required and then forward such requests to the default I/O target. If the filter driver completes a create request for a file object, it should set **AutoForwardCleanupClose** to **WdfFalse** in the file object configuration so that KMDF completes cleanup and close requests for the file object instead of forwarding them.

To handle file cleanup and close requests, a driver registers the **EvtFileCleanup** and **EvtFileClose** event callbacks. If a bus or function driver does not register such a callback, KMDF closes the file object and completes the request with **STATUS_SUCCESS**. In a filter driver that does not register cleanup and close callbacks, KMDF forwards these requests to the default I/O target unless the driver has explicitly set **AutoForwardCleanupClose** to **WdfFalse** in the file object configuration.

6.7.1.2 Read, Write, Device I/O Control, and Internal Device I/O Control Requests

For read, write, device I/O control, and internal device I/O control requests, the driver creates one or more queues and configures each queue to receive one or more types of I/O requests. When such a request arrives, the I/O request handler does the following:

- Determines whether the driver has configured a queue for this type of request. If not, the handler fails a read, write, device I/O control, or internal device I/O control request if this is a function or bus driver. If this is a filter driver, the handler passes such a request to the default I/O target.
- Determines whether the queue is accepting requests and the device is powered on. If both are true, the handler creates a **WDFREQUEST** object to represent the request and adds it to the queue. If the queue is not accepting requests, the handler fails the request.
- If the device is not in the DO state, notifies the Plug and Play/power handler to power up the device.
- Queues the request.

Figure 6.3 shows the flow of a read, write, device I/O control, or internal device I/O control request through the I/O request handler to the driver.

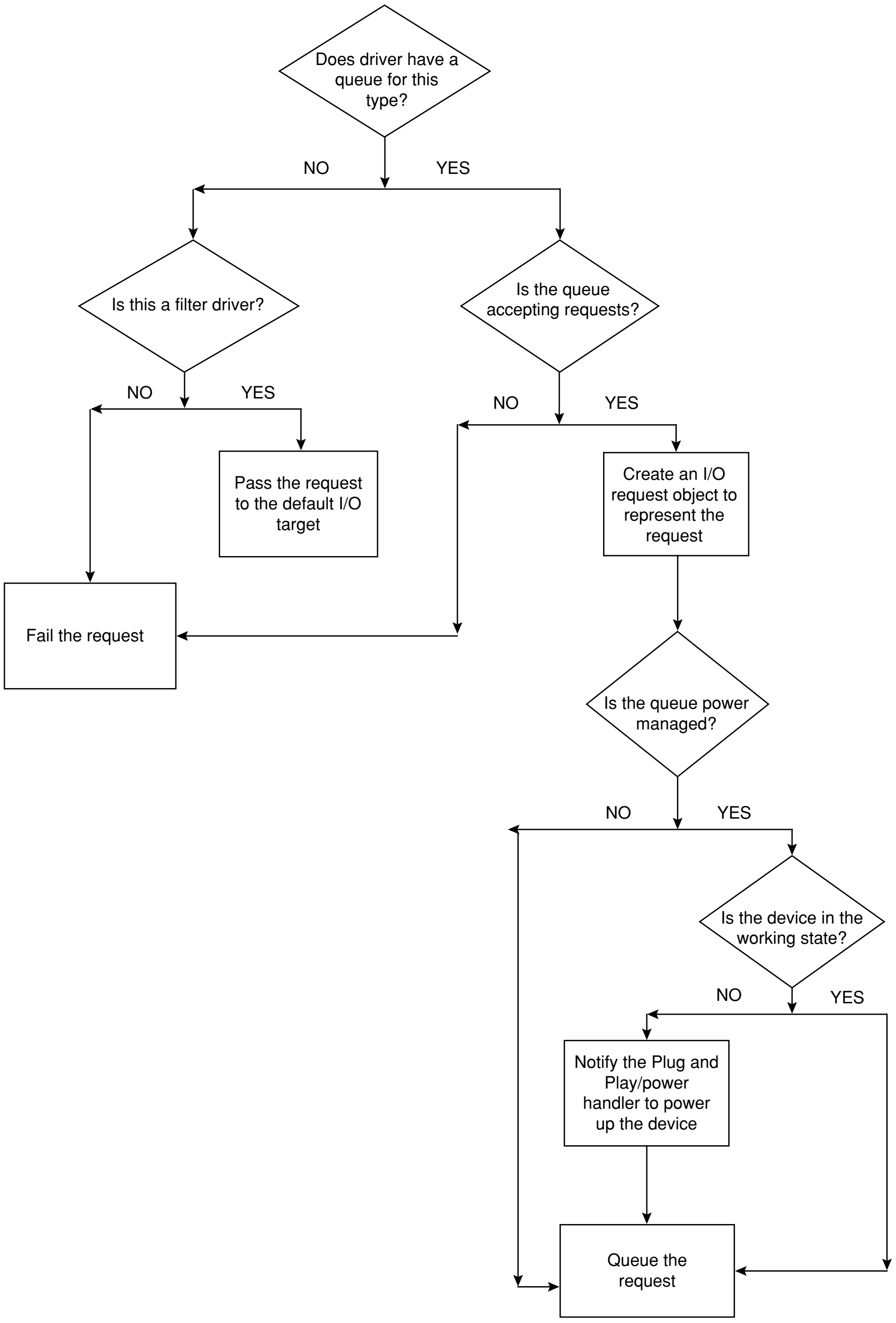
Does driver have a queue for this type?
NO
YES
Is this a filter driver?
Is the queue accepting requests?
NO
YES
NO
YES
Pass the request to the default I/O target
Create an I/O request object to represent the request
Fail the request
Is the queue power managed?
NO
YES
Is the device in the working state?
NO
YES
Notify the Plug and Play/power handler to power up the device
Queue the request

Figure 6.3 Flow of I/O Request Through I/O Request Handler

6.7.2 I/O Queues

A **WDFQueue** object represents a queue that presents requests from KMDF to the driver. A **WDFQUEUE** is more than just a list of pending requests; however, it tracks requests that are active in the driver, supports request cancellation, manages the concurrency of requests, and can optionally synchronize calls to the driver's I/O event callback functions.

A driver typically creates one or more queues, each of which can accept one or more types of requests. The driver configures the queues when it creates them. For each queue, the driver can specify

- The types of requests that are placed in the queue.
- The event callback functions that are registered to handle I/O requests from the queue.
- The power management options for the queue.
- The dispatch method for the queue, which determines the number of requests that are serviced at a given time.
- Whether the queue accepts requests that have a zero-length buffer.

A driver can have any number of queues, and they can all be configured differently. For example, a driver might have a parallel queue for read requests and a sequential queue for write requests.

While a request is in a queue and has not yet been presented to the driver, the queue is considered the "owner" of the request. After the request has been dispatched to the driver, it is "owned" by the driver and is considered an **in-flight** request internally, and each **WDFQUEUE** object keeps track of which requests it owns and which requests are pending. A driver can forward a request from one queue to another by calling a method on the request object.

6.7.2.1 Queues and Power Management

KMDF provides rich control of queues. The framework can manage the queues for the driver, or the driver can manage queues on its own. Power management is configurable on a per-queue basis. A driver can use both power-managed and non-power-managed queues and can sort requests based on the requirements for its power model.

By default, queues for FDOs and PDOs are power managed, which means that the state of the queue can trigger power-management activities. Such queues have several advantages:

- If an I/O request arrives while the system is in the working state (SO) but the device is not, KMDF notifies the Plug and Play/power handler so that it can restore device power.
- When a queue becomes empty, KMDF notifies the Plug and Play/power handler so that it can track device usage through its idle timer.
- If the device power state begins to change while the driver "owns" an I/O request, KMDF can notify the driver through the **EvtIoStop** callback. The driver must complete, cancel, or acknowledge all the I/O requests that it owns before the device can leave the working state.

For power-managed queues, KMDF pauses the delivery of requests when the device leaves the working state (DO) and resumes delivery when the device returns to the working state. Although delivery stops while the queue is paused, queuing does not. If KMDF receives a request while the queue is paused, KMDF adds the request to the queue for delivery after the queue resumes. If an I/O request arrives while the device is idle and the system is in the working state, KMDF returns the device to the working state so that it can handle the request. If an I/O request arrives while the system is transitioning to a sleep state, however, KMDF does not return the device to the working state until the system returns to the working state.

For requests to be delivered, both the driver and the device power state must allow processing. The driver can pause delivery manually by calling **WdfIoQueueStop** and resume delivery by calling **WdfIoQueueStart**.

If a queue is not power managed, the state of the queue has no effect on power management, and conversely, KMDF delivers requests to the driver any time the system is in the working state, regardless of the power state of the device. KMDF does not start an idle timer when the queue becomes empty, and it does not power up a sleeping device when I/O arrives for the queue.

Drivers should use non-power-managed queues to hold requests that the driver can handle even while its device is not in the working state.

6.7.2.2 Dispatch Type

A queue's dispatch type determines how and when I/O requests are delivered to the driver and, as a result, whether multiple I/O requests from a queue are active in the driver at one time. Drivers can control the concurrency of **in-flight** requests by configuring the dispatching method for their queues. KMDF supports three dispatch types:

- **Sequential—**A queue that is configured for sequential dispatching delivers I/O requests to the driver one at a time. The queue does not deliver another request to the driver until the previous request has been completed. (Sequential dispatching is similar to the start-I/O technique in WDM.)
- **Parallel—**A queue that is configured for parallel dispatching delivers I/O requests to the driver as soon as possible, whether or not another request is already active in the driver.
- **Manual—**A queue that is configured for manual dispatching does not deliver I/O requests to the driver. Instead, the driver retrieves requests at its own pace by calling a method on the queue.

The dispatch type controls only the number of requests that are active within a driver at one time. It has no effect on whether the queue's I/O event callbacks are invoked sequentially or concurrently; instead, the concurrency of callbacks is controlled by the synchronization scope of the device object. Even if the synchronization scope for a parallel queue does not allow concurrent callbacks, the queue nevertheless might have many **in-flight** requests.

All I/O requests that a driver receives from a queue are inherently asynchronous. The driver can complete the request within the event callback or sometime later, after returning from the callback.

6.7.3 I/O Request Objects

The **WDFREQUEST** object is the KMDF representation of an IRP. When an I/O request arrives, the I/O handler creates a **WDFREQUEST** object, queues the object, and eventually passes the object to the driver in its I/O callback function.

The properties of the **WDFREQUEST** object represent the fields of the IRP. The object also contains additional information. Like all other KMDF objects, the **WDFREQUEST** object has a reference count and can have its own object context area. When the driver completes the I/O request that the object represents, KMDF automatically frees the object and any child resources such as associated memory buffers or memory descriptor lists (MDLs). After the driver has called **WdfRequest-Complete**, the driver must not attempt to access the handle to the **WDFREQUEST** object or any of its child resources. A driver can create its own **WDFREQUEST** objects to request I/O from another device or to split an I/O request into multiple, smaller requests before completing it.

6.7.4 Retrieving Buffers from I/O Requests

The **WDFMEMORY** object encapsulates the I/O buffers that are supplied for an I/O request. To enable device drivers to handle complicated requests with widely scattered buffers, any number of **WDFMEMORY** objects may be associated with a **WDFREQUEST**.

The **WDFMEMORY** object represents a buffer that the framework manages. The object can be used to copy memory to and from the driver and the buffer represented by the **WDFMEMORY** handle. In addition, the driver can use the underlying buffer pointer and its length for complex access, such as casting to a known data structure.

Like other KMDF objects, **WDFMEMORY** objects have reference counts and persist until all references to them have been removed. The buffer that underlies the **WDFMEMORY** object, however, might not be "owned" by the object itself. For example, if the issuer of the I/O request allocated the buffer or if the driver called **WdfMemoryCreatePreallocated** to assign an existing buffer to the object, the **WDFMEMORY** object does not "own" the buffer. In this case, the buffer pointer becomes invalid when the associated I/O request has been completed, even if the **WDFMEMORY** object still exists.

Each **WDFMEMORY** object contains the length of the buffer that it represents. KMDF methods that copy data to and from the buffer validate the length of every transfer to prevent buffer over- and underruns, which can result in corrupt data or security breaches.

Depending on the type of I/O that the device and driver support, the underlying buffer might be any of the following:

- For buffered I/O, a system-allocated buffer from the nonpaged pool.
- For direct I/O, a system-allocated MDL that points to the physical pages for DMA.
- For neither buffered nor direct I/O, an unmapped and unverified user mode memory address.

The **WDFMEMORY** object supports methods that return each type of buffer from the object and methods to read and write the buffers. For device I/O control requests (IOCTLs), KMDF provides methods to probe and lock user mode buffers. The driver must be running in the context of the process that sent the I/O request to probe and lock a user mode buffer, so KMDF also defines a callback that drivers can register to be called in the context of the sending component.

Each **WDFMEMORY** object also controls access to the buffer and allows the driver to write only to buffers that support I/O from the device to the buffer. A buffer that is used to receive data from the device (as in a read request) is writable. The **WDFMEMORY** object does not allow write access to a buffer that only supplies data (as in a write request).

6.7.5 I/O Targets

Drivers send I/O requests by creating or reusing an I/O request object, creating an I/O target, and sending the request to the target. Drivers can send requests either synchronously or asynchronously. A driver can specify a time-out value for either type of request.

An I/O target represents a device object to which an I/O request is directed. If a driver cannot complete an I/O request by itself, it typically forwards the request to an I/O target. An I/O target can be a KMDF driver, a WDM driver, or any other Kernel Mode Driver.

Before a driver forwards an existing I/O request or sends a new request, it must create a **WDFIOTARGET** object to represent either a local or remote target for the I/O request. The local I/O target is the next lower driver in the device stack and is the default target or a filter or FDO device object. A remote I/O target is any other driver that might be the target of an I/O request. A driver might use a remote I/O target if it requires data from another device to complete an I/O request. A function driver might also use a remote I/O target to send a device I/O control request to its bus driver. In this case, the I/O request originates with the function driver itself, rather than originating with some other process.

The **WDFIOTARGET** object formats I/O requests to send to other drivers, handles changes in device state, and defines callbacks through which a driver can request notification about target device removal. A driver can call methods on the **WDFIOTARGET** to

- Open a device object or device stack by name.
- Format read, write, and device I/O control requests to send to the target. Some types of targets, such as **WDFUSBDEVICE** and **WDFUSBPIPE**, can format bus-specific requests in addition to the standard request types.

- Send read, write, and device I/O control requests synchronously or asynchronously.
- Determine the Plug and Play state of the target.

Internally, KMDF calls **IoCallDriver** to send the request. It takes out a reference on the **WDFREQUEST** object to prevent the freeing of associated resources while the request is pending for the target device object.

The **WDFIOTARGET** object racks queued and sent requests and can cancel them when the state of the target device or of the issuing driver changes. From the driver's perspective, the I/O target object behaves like a cancel-safe queue that retains forwarded requests until KMDF can deliver them. KMDF does not free the **WDFIOTARGET** object until all the I/O requests that have been sent to it are complete.

By default, KMDF sends a request only when the target is in the proper state to receive it. However, a driver can also request that KMDF ignore the state of the target and send the request anyway. If the target device has been stopped (but not removed), KMDF queues the request to send later after the target device resumes. If the issuing driver specifies a time-out value, the timer starts when the request is added to the queue.

If the device that is associated with a remote I/O target is removed, KMDF stops and closes the I/O target object, but does not notify the driver unless the driver has registered an **EvtIoTargetXxx** callback. If the driver must perform any special processing of I/O requests that it sent to the I/O target, it should register one or more such callbacks. When the removal of the target device is queried, canceled, or completed, KMDF calls the corresponding functions and then processes the target state changes on its own.

For local I/O targets, no such callbacks are defined. Because the driver and the target device are in the same device stack, the driver is notified of device removal requests through its Plug and Play and power management callbacks.

6.7.6 Creating Buffers for I/O Requests

Drivers that issue I/O requests must supply buffers for the results of those requests. The buffers in a synchronous request can be allocated from any type of memory, such as the nonpaged pool or an MDL, as well as a **WDFMEMORY** object. Asynchronous requests must use

WDFMEMORY object so that KMDF can ensure that the buffers persist until the I/O request has completed back to the issuing driver.

If the driver uses a **WDFMEMORY** object, the I/O target object takes out a reference on the **WDFMEMORY** object when it formats the object to send to the I/O target. The target object retains this reference until one of the following occurs:

- The request has been completed.
- The driver reformats the **WDFREQUEST** object.
- The driver calls **WdfRequestReuse** to send a request to another target.

A driver can retrieve a **WDFMEMORY** object from an incoming **WDFREQUEST** and reuse it later in a new request to a different target. However, if the driver has not yet completed the original request, the original I/O target still has a reference on the **WDFMEMORY** object. To avoid a bug check, the driver must call **WdfRequestReuse** in its I/O completion routine before it completes the original request.

6.7.7 Canceled and Suspended Requests

Windows I/O is inherently asynchronous. The system can request that a driver stop processing an I/O request at any time for many reasons, of which these are the most common:

- The thread or process that issued the request cancels it or exits.
- A system Plug and Play or power event such as hibernation occurs.
- The device is being, or has been, removed.

The actions that a driver takes to stop processing an I/O request depend on the reason for suspension or cancellation. In general, the driver can either cancel the request or complete it with an error. In some situations, the system might request that a driver suspend (temporarily pause) processing; the system notifies the driver later when to resume processing.

To provide a good user experience, drivers should provide callbacks to handle cancellation and suspension of any I/O request that might take a long time to complete or that might not complete, such as a request for asynchronous input.

6.7.7.1 Request Cancellation

How KMDF proceeds to cancel an I/O request depends on whether the request has already been delivered to the target driver:

- If the request has never been delivered—either because KMDF has not yet queued it or because it is still in a queue—KMDF cancels or suspends it automatically. If the original IRP has been canceled, KMDF completes the request with a cancellation status.
- If the request has been delivered and then requeued, KMDF notifies the driver of cancellation only if the driver has registered an **EvtIoCanceledOnQueue** callback for the queue.

After a request has been delivered, it cannot be canceled unless the driver that owns it explicitly marks it cancelable by calling the **WdfRequest-MarkCancelable** method on the request and registering a cancellation callback (**EvtRequestCancel**) for the request.

A driver should mark a request cancelable and register an I/O cancellation callback if either of the following is true:

- The request involves a long-term operation.
- The request might never succeed; for example, the request is waiting for synchronous input.

In the **EvtRequestCancel** callback, the driver must perform any tasks that are required to cancel the request, such as stopping any device I/O operations that are in progress and canceling any related requests that it has already forwarded to an I/O target. Eventually, the driver must complete the request with the status **STATUS_CANCELLED**.

Requests that are marked cancelable cannot be forwarded to another queue. Before requeuing a request, the driver must first make it non-cancelable by calling **WdfRequestUnmarkCancelable**. After the request has been added to the new queue, KMDF once again considers it cancelable until that queue dispatches it to the driver.

If the driver does not mark a request cancelable, it can call **WdfRequestIsCanceled** to determine whether the I/O manager or original requester has attempted to cancel the request. A driver that processes data on a periodic basis might use this approach. For example, a driver involved in image processing might complete a transfer request in small chunks and poll for cancellation after processing each chunk. In this case, the driver supports cancellation of the I/O request, but only after each discrete chunk of processing is complete. If the driver determines that the request has been canceled, it performs any required cleanup and completes the request with the status **STATUS_CANCELLED**.

6.7.7.2 Request Suspension

When the system transitions to a sleep state—typically because the user has requested hibernation or closed the lid on a laptop—a driver can complete, requeue, or continue to hold any **in-flight** requests. KMDF notifies the driver of the impending power change by calling the **EvtIoStop** callback for each **in-flight** request. Each call includes flags that indicate the reason for stopping the queue and whether the I/O request is currently cancelable.

Depending on the value of the flags, the driver can complete the request, requeue the request, acknowledge the event but continue to hold the request, or ignore the event if the current request will complete in a timely manner. If the queue is stopping because the device is being removed (either by an orderly removal or a surprise removal), the driver must complete the request immediately.

Drivers should handle **EvtIoStop** events for any request that might take a long time to complete or that might not complete, such as a request for asynchronous input. Handling **EvtIoStop** provides a good user experience for laptops and other power-managed systems.

6.7.8 Completing I/O Requests

To complete an I/O request, a driver calls **WdfRequestComplete**. In response, KMDF completes the underlying IRP and then deletes the **WDFREQUEST** object and any child objects. If the driver has set an **EvtCleanupCallback** for the **WDFREQUEST** object, KMDF invokes the callback before completing the underlying IRP, so that the IRP itself is still valid when the callback runs.

After **WdfRequestComplete** returns, the **WDFREQUEST** object's handle is invalid and its resources have been released. The driver must not attempt to access the handle or any of its resources, such as parameters and memory buffers that were passed in the request.

If the request was dispatched from a sequential queue, the driver's call to complete the IRP might cause KMDF to deliver the next request in the queue. (If the queue is configured for parallel dispatching, KMDF can deliver another request at any time.) If the driver holds any locks while it calls **WdfRequestComplete**, it must ensure that its event callbacks for the queue do not use the same locks because a deadlock might occur. In practice, this is difficult to ensure, so the best practice is not to call **WdfRequestComplete** while holding a lock.

6.7.9 Self-Managed I/O

Although the I/O support that is built into KMDF is recommended for most drivers, some drivers have I/O paths that do not pass through queues or are not subject to power management. KMDF provides self-managed I/O features for this purpose. For example, the PCIDRV sample uses self-managed I/O callbacks to start and stop a watchdog timer DPC.

The self-managed I/O callbacks correspond directly to WDM Plug and Play and power management state changes. These routines are called with a handle to the device object and no other parameters. If a driver registers these callbacks, KMDF calls them at the designated times so that the driver can perform whatever actions it requires.

6.7.10 Accessing IRPs and WDM Structures

KMDF includes a mechanism nicknamed "the great escape" through which a driver can access the underlying WDM structures and the I/O request packet as it was delivered from the operating system. Although this mechanism exposes the driver to all the complexity of the WDM model, it can often be useful in converting a WDM driver to KMDF, such as processing for some types of IRPs. Such drivers can use KMDF for most features but can rely on the "great escape" to gain access to the WDM features that they require.

To use the "great escape," a driver calls **WdfDeviceInit-AssignWdmIrpPreprocessCallback** to register an **EvtDeviceWdm-IrpPreprocess** event callback function for an IRP major function code. When KMDF receives an IRP with that function code, it invokes the callback. The driver must then handle the request just as a WDM driver would, by using I/O manager functions such as **IoCallDriver** to forward the request and **IoCompleteRequest** to complete it. The Serial driver sample shows how to use this feature.

In addition to the "great escape," KMDF provides methods with which a driver can access the WDM objects that the KMDF objects represent. For example, a driver can access the IRP that underlies a **WDFREQUEST** object, the WDM device object that underlies a **WDFDEVICE** object, and so forth.

CHAPTER 7

Plug and Play and Power Management

The Windows Driver foundation (WDF) implements a fully integrated model for Plug and Play and power management. In this chapter, we will cover the Kernel Mode Driver Framework (KMDF) guidelines for implementing Plug and Play and power management. The model provides intelligent defaults so that some drivers do not require any code to support simple Plug and Play or power management.

7.1 Plug and Play and Power Management Overview

KMDF implements integrated Plug and Play and power management support as an internal state machine. An event is associated with the transition to each state, and a driver can supply callback routines that are invoked at each such state change.

If you are familiar with WDM drivers, you probably remember that any time the system power state changes, the WDM driver must determine the correct power state for its device and then issue power management requests to put the device in that state at the appropriate time. The KMDF state machine automatically handles the translation of system power events to device power events. For example, KMDF notifies the driver to

- Transition the device to low power when the system hibernates or goes to sleep.
- Enable the device's wake signal so that it can be triggered while the system is running, if the device is idle.
- Enable the device's wake signal so that it can be triggered while the system is in a sleep state.

KMDF automatically provides for the correct behavior in device parent/child relationships. If both a parent and a child device are powered down and the child must power up, KMDF automatically returns the parent to full power and then powers up the child.

To manage idle devices, the KMDF state machine notifies the driver to remove the device from the working state and put it in the designated low-power state when the device is idle and to return the device to the working state when there are requests to process.

To accomplish these power transitions, a driver includes a set of callback routines. These routines are called in a defined order, and each conforms to a "contract" so that both the device and the system are guaranteed to be in a particular state when the driver is called to perform an action. This support makes it much easier for drivers to power down idle devices. The driver simply sets an appropriate time-out value and low-power state for its device and notifies KMDF of these values; KMDF calls the driver to power down the device at the correct times.

In addition, requests received by the framework and not yet delivered to the device driver can affect the power state of the device. If the driver has not configured a queue for power management, the framework can automatically restore device power before it delivers the request to the driver. It can also automatically stop and start the queue in response to Plug and Play and power events.

Finally, the driver that manages power policy for the device can specify whether a user can control both the behavior of the device while it is idle and the capability of the device to wake up the system. All the driver must do is specify the appropriate enumerator value when it initializes certain power policy settings. KMDF enables the necessary property sheet through Windows Management Instrumentation (WMI), and Device Manager displays it.

7.2 Device Enumeration and Startup

To prepare the device for operation, KMDF calls the driver's callback routines in a fixed sequence. The sequence varies somewhat depending on the driver's role in the device stack.

7.2.1 Startup Sequence for a Function or Filter Device Object

The following example shows the callbacks for an FDO or filter DO that is involved in bringing a device to the fully operational state, starting from the Device Inserted state at the bottom of the figure.

	Device Operational
Enable self-managed I/O, if driver supports it	**EvtDeviceSelfManagedIoInit** or **EvtDeviceSelfManagedIoRestart**
Start power-managed queues (called only if **EvtIoStop** was previously called during power-down)	**EvtIoResume**
Disarm wake signal, if it was armed	**EvtDeviceDisarmWakeFromSx** **EvtDeviceDisarmWakeFromSO** (called only during power-up; not called during resource rebalance)
Enable DMA, if driver supports it	**EvtDmaEnablerSelfManagedIoStart** **EvtDmaEnablerEnable** **EvtDmaEnableFill**
Connect interrupts	**EvtDeviceDOEntryPostInterruptsEnabled** **EvtInterruptEnable**
Notify driver of state change	**EvtDeviceDOEntry**
	Restart from here if device is in low-power state
Prepare hardware for power	**EvtDevicePrepareHardware**
Change resource requirements	**EvtDeviceRemoveAddedResources** **EvtDeviceFilterAddResourceRequirements** **EvtDeviceFilterRemoveResourceRequirements**
	Restart from here if rebalancing resources
Create device object	**EvtDriverDeviceAdd**
	Device Inserted

The spaces between the entries in the preceding example mark the steps that are involved in starting a device. The column on the left side of the figure describes the step, and the column on the right lists the event callbacks that accomplish it.

At the bottom of the figure, the device is not present on the system. When the user inserts it, KMDF begins by calling the driver's **EvtDriverDeviceAdd** callback so that the driver can create a device object to represent the device. KMDF continues calling the driver's callback routines by progressing up through the sequence until the device is operational. Remember that KMDF invokes the event callbacks in bottom-up order as shown in the figure, so **EvtDeviceFilterRemoveResourceRequirements** is called before **EvtDeviceFilterAddResourceRequirements**, and so forth.

If the device was stopped to rebalance resources or was physically present but not in the working state, not all of the steps are required as the figure shows.

7.2.2 Startup Sequence for a Physical Device Object

The following shows the callbacks for a bus driver (PDO) that are involved in bringing a device to the fully operational state, starting Device Inserted state at the bottom of the figure.

Device Operational

Enable wake signal, if a wake request from the previous power-down is still pending	**EvtDeviceEnableWakeAtBus**
Enable self-managed I/O, if driver supports it	**EvtDeviceSelfManagedIoInit** or **EvtDeviceSelfManagedIoRestart**
Start power-managed queues	**EvtIoResume** (called only if **EvtIoStop** was called during power-down)
Enable DMA, if driver supports it	**EvtDmaEnablerSelfManagedIoStart** **EvtDmaEnablerEnable** **EvtDmaEnablerFill**
Connect interrupts	**EvtDeviceDOEntryPostInterruptsEnabled** **EvtInterruptEnable**

Notify driver of state change	**EvtDeviceD0Entry**
Disable wake signal, if it was	**EvtDeviceDisableWakeAtBus** (called only during power-up; not called during resource rebalance)
Restart from here if device is in low-power state	
Prepare hardware for power	**EvtDevicePrepareHardware**
Restart from here if rebalancing resources or if device remained physically present after logical removal	
Create device object	**EvtDriverDeviceAdd**
Report resource requirements	**EvtDeviceResourceRequirementsQuery** **EvtDeviceResourcesQuery**
Enumerate child devices	**EvtChildListCreateDevice**
Device Inserted	

KMDF does not physically delete a PDO until the corresponding device is physically removed from the system. For example, if a user disables the device in Device Manager but does not physically remove it, KMDF retains its device object. Thus, the three steps at the bottom of the preceding example occur only during Plug and Play enumeration—that is, during initial boot or when the user plugs in a new device.

If the device was previously disabled but not physically removed, KMDF starts by calling the **EvtDevicePrepareHardware** callback.

7.2.3 Device Power-Down and Removal

KMDF can remove a device from the operational state for several reasons:

- To put the device in a low-power state because it is idle or the system is entering a sleep state.
- To rebalance resources.
- To remove the device after the user has requested an orderly removal.
- To disable the device in response to the user's request in Device Manager.

As in enumeration and power-up, the sequence of callbacks depends on the driver's role in device management.

7.2.3.1 Power-Down and Removal Sequence for a Function or Filter Driver Object

The following shows the sequence of callbacks that are involved in power-down and removal for an FDO or filter DO. The sequence starts at the top of the figure with an operational device that is in the working power state (DO).

Device Operational

Suspend self-managed I/O, if driver supports it	**EvtDeviceSelfManagedIoSuspend**
Stop power-managed queues	**EvtIoStop**
Arm wake signal, if driver supports it	**EvtDeviceArmWakeFromSx** **EvtDeviceArmWakeFromSO** (called only during transitions to low power, not during resource rebalance or device removal)
Disable DMA, if driver supports it	**EvtDmaEnablerSelfManagedIoStop** **EvtDmaEnablerDisable** **EvtDmaEnableFlush**
Disconnect interrupts	**EvtDeviceDOExitPreInterruptsDisabled** **EvtInterruptDisable**
Notify driver of state change	**EvtDeviceDOExit**

Stop here if transitioning to low-power state

Release hardware	**EvtDeviceReleaseHardware**

Stop here if rebalancing resources

Purge power-managed queues	**EvtIoStop**
Flush I/O buffers, if driver supports self-managed I/O	**EvtDeviceSelfManagedIoFlush**
Purge non-power-managed queues	**EvtIoStop**
Clean up I/O buffers, if driver supports self-managed I/O	**EvtDeviceSelfManagedIoCleanup**
Delete device object's context area	**EvtCleanupContext** **EvtDestroyContext**

Device Removed

As the preceding example shows, the KMDF power-down and removal sequence involves calling the corresponding "undo" callbacks in the reverse order from which KMDF called the functions that are involved in making the device operational.

7.2.3.2 Power-Down and Removal Sequence for a Physical Device Object

The following example shows the callbacks involved in power-down and removal for a PDO.

Device Operational	
Enable wake signal, if driver supports it (called only during transitions to lower power, not during resource rebalance or device removal)	**EvtDeviceEnableWakeAtBus**
Suspend self-managed I/O, if driver supports it	**EvtDeviceSelfManagedIoSuspend**
Stop power-managed queues	**EvtIoStop**
Disable DMA, if driver supports it	**EvtDmaEnablerSelfManagedIoStop** **EvtDmaEnablerDisable** **EvtDmaEnableFlush**
Disconnect interrupts	**EvtDeviceDOExitPreInterruptsDisabled** **EvtInterruptDisable**
Notify driver of state change	**EvtDeviceDOExit**
Disable wake signal, if it is enabled	**EvtDeviceDisableWakeAtBus** (called only during device removal)
Stop here if transitioning to low-power state	
Release hardware	**EvtDeviceReleaseHardware**
Stop here if rebalancing resources	
Purge power-managed I/O queues	**EvtIoStop**

(continues)

Flush I/O buffers, if driver supports	**EvtDeviceSelfManagedIoFlush**
Stop here if device is still physically present	
Purge non-power-managed I/O	**EvtIoStop**
Clean up I/O buffers, if driver supports self-managed I/O	**EvtDeviceSelfManagedIoCleanup**
Delete device object's context area	**EvtCleanupContext** **EvtDestroyContext**
Device Physically Removed	

KMDF does not physically delete the PDO until the device is physically removed from the system. For example, if a user disables the device in Device Manager or uses the Safely Remove Hardware utility to stop the device but does not physically remove it, KMDF retains the PDO. If the device is later re-enabled, KMDF uses the **EvtDevicePrepareHardware** callback, as previously shown in section 7.2.2, "Startup Sequence for a Physical Device Object."

7.2.3.3 Surprise-Removal Sequence

If the user removes the device without warning, by simply unplugging it without using Device Manager or the Safely Remove Hardware utility, the device is considered *surprise removed*. When this occurs, KMDF follows a slightly different removal sequence. It also follows the surprise-removal sequence if another driver calls **IoInvalidateDeviceState** on the device, even if the device is still physically present.

In the surprise-removal sequence, KMDF calls the **EvtDeviceSurpriseRemoval** callback before calling any of the other callbacks in the removal sequence. When the sequence is complete, KMDF destroys the device object.

Drivers for all removable devices must ensure that the callbacks in both the shutdown and startup paths can handle failure, particularly failures caused by the removal of the hardware. Any attempts to access the hardware should not wait indefinitely, but should be subject to time-outs or a watchdog timer.

The following example shows the surprise-removal sequence.

Device Removal

Notify driver that device has been surprise removed	**EvtDeviceSurpriseRemoval**
Suspend self-managed I/O (called only if the device was in the working state at removal)	**EvtDeviceSelfManagedIoSuspend**
Stop power-managed queues (called only if the device was in the working state at removal)	**EvtIoStop**
Disable DMA, if driver supports it (called only if the device was in the working state at removal)	**EvtDmaEnablerSelfManagedIoStop** **EvtDmaEnablerDisable** **EvtDmaEnablerFlush**
Disconnect interrupts (called only if the device was in the working state at removal)	**EvtDeviceD0ExitPreInterruptsDisabled** **EvtInterruptDisable**
Notify driver of state change (called only if the device was in the working state at removal)	**EvtDeviceD0Exit**
Release hardware	**EvtDeviceReleaseHardware**
Purge power-managed queues	**EvtIoStop**
Flush and clean up I/O buffers, if driver supports self-managed I/O	**EvtDeviceSelfManagedIoFlush**
Purge non-power-managed queues	**EvtIoStop**
Clean up I/O buffers, if driver supports self-managed I/O	**EvtDeviceSelfManagedIoCleanup**
Delete device object's context area	**EvtCleanupContext** **EvtDestroyContext**

Removal Processing Complete

If the device was not in the working state when it was removed, KMDF calls the **EvtDeviceReleaseHardware** event callback immediately after **EvtDeviceSurpriseRemoval**. It omits the intervening steps, which were already performed when the device exited from the working state.

7.3 WMI Request Handler

WMI provides a way for drivers to export information to other components. Drivers typically use WMI to enable the following:

- User mode applications to query and set device-related information, such as time-out values.
- An administrator with the necessary privileges to control a device by running an application on a remote system.

A driver that supports WMI registers as a provider of information and registers one or more instances of that information. Each WMI provider is associated with a particular Globally Unique Identifier (GUID). Another component can register with the same GUID to consume the data from the instances. User mode components request WMI instance data by calling COM functions, which the system translates into **IRP_MJ_SYSTEM_CONTROL** requests and sends to the target providers.

KMDF supports WMI requests through its WMI request handler, which provides the following features for drivers:

- A default WMI implementation. Drivers that do not provide WMI data are not required to register as WMI providers; KMDF handles all **IRP_MJ_SYSTEM_CONTROL** requests.
- Callbacks on individual instances, rather than just at the device object level, so that different instances can behave differently.
- Validation of buffer sizes to ensure that buffers that are used in WMI queries meet the size requirements of the associated provider and instance.

The default WMI implementation includes support for the check boxes on the Power Management tab of Device Manager. These check boxes enable a user to control whether the device can wake the system and whether the system can power down the device when it is idle. WDM drivers must include code to support the WMI controls that map to these check boxes, but KMDF drivers do not require such code. If the driver enables this feature in its power policy options, KMDF handles these requests automatically.

The driver enables buffer size validation when it configures a WMI provider object (**WDFWMIPROVIDER**). In the **WDF_WMI_PROVIDER_CONFIG** structure, the driver can specify the minimum size of the buffer that is required for the provider's **EvtWmiInstance-QueryInstance** and **EvtWmiInstanceSetInstance** callbacks. If the driver specifies such a value, KMDF validates the buffer size when the **IRP_MJ_SYSTEM_CONTROL** request arrives and calls the callbacks only if the supplied buffer is large enough. If the driver does not configure a buffer size—because the instance size is either dynamic or is not available when the provider is created—the driver should specify zero for this field and the callbacks themselves should validate the buffer size.

When KMDF receives an **IRP_MJ_SYSTEM_CONTROL** request that is targeted at a KMDF driver, it proceeds as follows:

- If the driver has registered as a WMI provider and registered one or more instances, the WMI handler invokes the callbacks for those instances as appropriate.
- If the driver has not registered any WMI instances, the WMI handler responds to the request by providing the requested data (if it can), passing the request to the next lower driver, or failing the request.

Like all KMDF objects, WMI instance objects (**WDFWMIINSTANCE**) have a context area. A driver can use the context area of a **WDFWMIINSTANCE** object as a source of read-only data, thus enabling easy data collection with minimal effort. A driver can delete **WDFWMIINSTANCE** objects any time after their creation.

WMI callbacks are not synchronized with the Plug and Play and power management state of the device. Therefore, when WMI events occur, KMDF calls a driver's WMI callbacks even if the device is not in the working state.

7.4 Synchronization Issues

Because Windows is a pre-emptive, multitasking operating system, multiple threads can try to access shared data structures or resources concurrently and multiple driver routines can run concurrently. To ensure data integrity, all drivers must synchronize access to shared data structures. Correctly implementing such synchronization can be difficult in WDM drivers.

For KMDF drivers, ensuring proper synchronization requires attention to several areas:

- The number of concurrently active requests that are dispatched from a particular queue.
- The number of concurrently active callbacks for a particular object.
- The driver utility functions that access object-specific data.
- The IRQL at which an object's callbacks run.

The dispatch method for an I/O queue controls the number of requests from the queue that can be concurrently active in the driver, as described previously in the section "Dispatch Type" in both Chapters 3 and 6. Limiting concurrent requests does not, however, resolve all potential synchronization issues. Concurrently active callbacks on the same object might require access to shared object-specific data, such as the information that is stored in the object context area. Similarly, driver utility functions might share object-specific data with callbacks. Furthermore, a driver must be aware of the **IRQL** at which its callbacks can be invoked. At **DISPATCH_LEVEL** and above, drivers must not access pageable data and thread pre-emption does not occur.

KMDF simplifies synchronization for driver by providing automatic synchronization of many callbacks. Calls to most PDO, FDO, Plug and Play, and power event callback functions are synchronized so that only one such callback function is invoked at a time for each device. These callback functions are called at **IRQL PASSIVE_LEVEL**. Note, however, that calls to the **EvtDeviceSurpriseRemoval**, **EvtDeviceQueryRemove**, and **EvtDeviceQueryStop** callbacks are not synchronized with the other callbacks and so occur while the device is changing power state or is not in the working state.

For other types of callbacks—primarily I/O related callbacks—the driver can specify the synchronization scope (degree of concurrency) and the maximum execution level (**IRQL**).

KMDF provides the following configurable synchronization features:

- Synchronization scope
- Execution level
- Locks

Although implementing synchronization is much less complicated in KMDF drivers than in WDM drivers, you should nevertheless be familiar with the basics of Windows IRQL, synchronization, and locking.

7.4.1 Synchronization Scope

KMDF provides configurable concurrency control, called *synchronization scope*, for the callbacks of several types of objects. An object's synchronization scope determines whether KMDF invokes certain event callbacks on the object concurrently.

KMDF define the following synchronization scopes:

- **Device scope** means that KMDF does not call certain I/O event callbacks concurrently for an individual device object or any file objects or queues that are children of the device object. Specifically, device scope applies to the following event callbacks: **EvtDeviceFileCreate**, **EvtFileCleanup**, **EvtFileClose**, **EvtIoDefault**, **EvtIoRead**, **EvtIoWrite**, **EvtIoDeviceControl**, **EvtIoInternalDeviceControl**, **EvtIoStop**, **EvtIoResume**, **EvtIoQueueState**, **EvtIoCanceledOnQueue**, and **EvtRequestCancel**.

  However, callbacks for different device objects that were created by the same driver object can be called concurrently. Internally, KMDF creates a *synchronization lock* for each device object. To implement device synchronization scope, KMDF acquires this lock before invoking any of the device callbacks.
- **Queue scope** means that KMDF does not call certain I/O callbacks concurrently on a per-queue basis. If a Kernel Mode Driver specifies queue scope for a device object, some callbacks for the device object and its queues can run concurrently. However, the following callbacks for an individual queue object are not called concurrently: **EvtIoDefault**, **EvtIoRead**, **EvtIoWrite**, **EvtIoDeviceControl**, **EvtIoInternalDeviceControl**, **EvtIoStop**, **EvtIoResume**, **EvtIoQueueState**, **EvtIoCanceledOnQueue**, and **EvtRequestCancel**. If the driver specifies queue scope, KMDF creates a synchronization lock for each queue object and acquires this lock before invoking any of the listed callbacks.
- **No scope** means that KMDF does not acquire any locks and can call any event callback concurrently with any other event callback. The driver must create and acquire all its own locks. By default,

KMDF uses no scope. A driver must "opt in" to synchronization for its objects by setting device scope explicitly.

Each KMDF object inherits its scope from its parent object (**WdfSynchronizationScopeInheritFromParent**). The parent of each **WDFDEVICE** object is the **WDFDRIVER** object, and the default value of the synchronization scope for the **WDFDRIVER** object is **WdfSynchronizationScopeNone**. Thus, a driver must explicitly set the synchronization scope on its objects to use frameworks synchronization.

A driver can change the scope by setting a value in the **WDR_OBJECT_ATTRIBUTES** structure when it creates the object. Because scope is inherited, a driver can easily set synchronization for most of its objects by setting the scope for the device object, which is the parent to most KMDF objects. (For the complete hierarchy, refer to Figure 6.1.)

For example, to set the concurrency for its I/O callback functions, a driver sets the **SynchronizationScope** in the **WDF_OBJECT_ATTRIBUTES** for the device object that is the parent to the I/O queues. If the driver sets device scope (**WdfSynchronizationScopeDevice**), KMDF calls only one I/O callback function at a time across all the queues. To use queue scope, the driver sets **WdfSynchronizationScopeQueue** for the device object and **WdfSynchronizationScopeInheritFromParent** for the queue object. Queue scope means that only one of the listed callback functions can be active for the queue at any time. A driver cannot set concurrency separately for each queue. Restricting the concurrency of I/O callbacks can help to manage access to shared data in the **WDFQUEUE** context memory.

By default, a file object inherits its scope from its parent device object. Attempting to set queue scope for a file object causes an error. Therefore, drivers that set queue scope for a device object must manually set the synchronization scope for any file objects that are its children. The best practice for file objects is to use no scope and to acquire locks in the event callback functions when they are required to synchronize access.

If a driver sets device scope for a file object, it must also set the passive execution level for the object, as described in the upcoming section "Execution Level." The reason is that the framework uses spin locks (which raise **IRQL** to **DISPATCH_LEVEL**) to synchronize access to objects with device scope. However, the **EvtDeviceFileCreate**, **EvtFileClose**, and **EvtFileCleanup** callbacks run in the caller's thread context and use pageable data, so they must be called at

PASSIVE_LEVEL. At **PASSIVE_LEVEL**, the framework uses a **FAST_MUTEX** instead of a spin lock for synchronization.

Interrupt objects are the children of device objects. KMDF acquires the interrupt object's spin lock at device interrupt request level (DIRQL) to synchronize calls to the **EvtInterruptEnable**, **EvtInterruptDisable**, and **EvtInterruptlsr** callbacks. A driver can also ensure that calls to its interrupt object's **EvtInterruptDpc** callback are serialized with other callbacks on the parent device object.

Deferred Procedure Call (DPC), timer, and work item objects can be children of device objects or of queue objects. To simplify a driver's implementation of callbacks for DPCs, timers, and work items, KMDF enables the driver to synchronize their callbacks with those of either the associated queue object or the device object (which might be the parent or the grandparent of the DPC, timer, or work item).

A driver sets callback synchronization on interrupt, DPC, timer, and work item objects by setting **AutomaticSerialization** in the object's configuration structure during object creation.

7.4.2 Execution Level

KMDF drivers can specify the maximum **IRQL** at which the callbacks for driver, device, file, and general objects are invoked. Like synchronization scope, execution level is an attribute that the driver can configure when it creates the object. KMDF supports the following execution levels:

- **Default execution level** indicates that the driver has placed no particular constraints on the **IRQL** at which the callbacks for the object can be invoked. For most objects, this is the default.
- **Passive execution level** (**WdfExecutionLevelPassive**) means that all event callbacks for the object occur at **PASSIVE_LEVEL**. If necessary, KMDF invokes the callback from a system worker thread. Drivers can set this level only for device and file object. Typically, a driver should set passive execution level only if the callbacks access pageable code or data or call other functions that must be called at **PASSIVE_LEVEL**.

  Callbacks for events on file objects (**WDFFILEOBJECT** type) are always called at **PASSIVE_LEVEL** because these functions must be able to access pageable code and data.

- **Dispatch execution level** (**WdfExecutionLevelDispatch**) means that KMDF can invoke the callbacks from any **IRQL** up to and including **DISPATCH_LEVEL**. This setting does not force all callbacks to occur at **DISPATCH_LEVEL**. However, if a callback requires synchronization, KMDF uses a spin lock, which raises **IRQL** to **DISPATCH_LEVEL**. Drivers can set dispatch execution level but nevertheless ensure that some tasks are performed at **PASSIVE_LEVEL** by using work items (**WDFWORKITEM** objects). Work item callbacks are always invoked at **PASSIVE_LEVEL** in the context of a system thread.

By default, an object inherits its execution level from its parent object. The default execution level for the **WDFDRIVER** object is **WdfExecutionLevelDispatch**.

7.4.3 Locks

In addition to internal synchronization, synchronization scope, and execution level, KMDF provides the following additional ways for a driver to synchronize operations:

- Acquire the lock that is associated with a device or queue object.
- Create and use additional, KMDF-defined, driver-created lock objects.

Driver code that runs outside an event callback sometimes must synchronize with code that runs inside an event callback. To accomplish this synchronization, KMDF provides methods (**WdfObjectAcquireLock** and **WdfObjectReleaseLock**) through which the driver can acquire and release the internal framework lock that is associated with a particular device or queue object.

Given the handle to a device or queue object, **WdfObjectAcquireLock** acquires the lock that protects that object. After acquiring the lock, the driver can safely access the object context data or properties and can perform other actions that affect the object. If the driver has set **WdfExecutionLevelPassive** for the object (or if the object has inherited this value from its parent), KMDF uses a **PASSIVE_LEVEL** synchronization primitive (a fast mutex) for the lock. If the object does not have this constraint, use of the lock raises **IRQL** to **DISPATCH_LEVEL** and, while the driver holds the lock, it must not touch pageable code or data or call functions that must run at **PASSIVE_LEVEL**.

KMDF also defines two types of lock objects:

- **Wait locks** (**WDFWAITLOCK**) synchronize access from code that runs at **IRQL PASSIVE_LEVEL** or **APC_LEVEL**. Such locks prevent thread suspension. Internally, KMDF implements wait locks by using kernel dispatcher events, so each wait lock is associated with an optional time-out value (as are the kernel dispatcher events). If the time-out value is zero, the driver can acquire the lock at **DISPATCH_LEVEL**.
- **Spin locks** (**WDFSPINLOCK**) synchronize access from code that runs at any **IRQL** up to **DISPATCH_LEVEL**. Because code that holds a spin lock runs at **DISPATCH_LEVEL**, it cannot take a page fault and therefore must not access any pageable data. The **WDFSPINLOCK** object keeps track of its acquisition history and ensures that deadlocks cannot occur. Internally, KMDF uses the system's spin lock mechanisms to implement spin lock objects.

As with all other KMDF objects, each instance of a lock object can have its own context area that holds lock-specific information.

Drivers that do not use the built-in frameworks locking (synchronization scope, execution level, and **AutomaticSerialization**) can implement their own locking schemes by using KMDF **wait** locks and spin locks. Drivers that use frameworks locking can use KMDF wait locks and spin locks to synchronize access to data that is not associated with a particular device or queue object. In general, drivers can rely on frameworks locking while communicating with their own hardware and calling within their own code. Drivers that communicate with other drivers generally must implement their own locking schemes.

7.4.4 Interaction of Synchronization Mechanisms

Synchronization scope and execution level interact because of the way in which KMDF implements synchronization. By default, KMDF uses spin locks, which raise **IRQL** to **DISPATCH_LEVEL**, to synchronize callbacks. Therefore, if the driver specifies device or queue synchronization scope, its device and queue callbacks must be able to run at **DISPATCH_LEVEL**.

If the driver sets the **WdfExecutionLevelPassive** constraint for a parent device or queue object, KMDF uses a fast mutex instead of a spin lock. In this case, however, KMDF cannot automatically synchronize callbacks for timer and DPC child objects (including the DPC object that is associated with the interrupt object) because DPC and timer callbacks, by definition, always run at **DISPATCH_LEVEL**. Trying to create any of these objects with **AutomaticSerialization** fails if the **WdfExecutionLevelPassive** constraint is set for the parent object.

Instead, the driver can synchronize the event callbacks for these objects by using a **WDFSPINLOCK** object. The driver acquires and releases the lock manually by the KMDF locking methods **WdfSpinLockAcquire** and **WdfSpinLockRelease**.

Alternatively, the driver can perform whatever processing is required within the DPC or timer callback and then queue a work item that is synchronized with the callbacks at **PASSIVE_LEVEL** to perform further detailed processing.

7.5 Security

KMDF is designed to enhance the creation of secure driver by providing:

- Safe defaults
- Parameter validation
- Counted Unicode strings
- Safe device naming techniques

7.5.1 Safe Defaults

Unless the driver specifies otherwise, KMDF provides access control lists (ACLs) that require Administrator privileges for access to any exposed driver constructs, such as names, device IDs, WMI management interfaces, and so forth. In addition, KMDF automatically handles I/O requests for which a driver has not registered by completing them with **STATUS_INVALID_DEVICE_REQUEST**.

7.5.2 Parameter Validation

One of the most common driver security problems involves improper handling of buffers in **IOCTL** requests, particularly requests that specify neither buffered nor direct I/O (**METHOD_NEITHER**). By default, KMDF does not grant drivers direct access to user mode buffer pointers, which is inherently unsafe. Instead, it provides methods for accessing the user mode buffer pointer that require probing and locking, and it provides methods to probe and lock the buffer for reading and writing.

All KMDF DDIs that require a buffer take a length parameter that specifies a required minimum buffer size. I/O buffers use the **WDFMEMORY** object, which provides data access methods that automatically validate length and determine whether the buffer permissions allow write access to the underlying memory.

7.5.3 Counted UNICODE Strings

To help prevent string handling errors, KMDF DDIs use only counted **PUNICCODE_STRING** values. To aid drivers in using and formatting **UNICODE_STRING** values, the safe string routines in **ntstrsafe.h** have been updated to take **PUNICODE_STRING** parameters.

7.5.4 Safe Device Naming Techniques

KMDF device objects do not have fixed names. KMDF sets **FILE_AUTOGENERATED_DEVICE_NAME** in the device's characteristics for PDOs, according to the WDM requirements.

KMDF also supports the creation and registration of device interfaces on all Plug and Play devices and manages device interfaces for its drivers. Whenever possible, you should use device interfaces instead of the older fixed name/symbolic link techniques.

However, if legacy applications require that a device has a name, KMDF enables you to name a device and to specify its security description definition language (SDDL). The SDDL controls which users can open the device.

By convention, a fixed device name is associated with a fixed symbolic link name (such as **\DosDevices\MyDeviceName**). KMDF supports the creation and management of a symbolic link and automatically deletes the link when the device is destroyed. KMDF also enables the creation of a symbolic link name for an unnamed Plug and Play device.

This page intentionally left blank

CHAPTER 8

Kernel Mode Installation and Build

In this chapter, we will cover the kernel mode installation and build approach.

Although KMDF supports a completely new device-driver interface (DDI) and programming model, the basic process of implementing and building a KMDF driver still has much in common with Windows Driver Model (WDM). If you are new to driver development, here are a few key points:

- Drivers are normally written in C. C++ can be used for driver development in only a very limited way. You can safely use some basic C++ features, but the object-oriented features of C++ produce generated code that is not guaranteed to work correctly in kernel mode.
- You can use a .cpp extension with the C++ compiler to compile driver code. The C++ compiler works fine with C code and provides better error detection and type safety than the C compiler.
- Include **Ntddk.h** and **Wdf.h**. These are standard header files that are used for all KMDF drivers.
- Drivers must be built with the **WDK** or **DDK** build tools. Microsoft Visual Studio is not designed to support driver development and can be used only in a limited way.

8.1 WDK Build Tools

KMDF drivers are built with the **WDK** build utility **build.exe.** This is a command line tool that is essentially identical to the tool that is used to build Windows itself. The build utility can be used for a variety of project types including user mode applications, but it must be used for drivers.

The build utility requires a number of supporting files. The following are required for any project:

- **Source code files**—A project must have at least one source code file (.c or .cpp) and typically one or more header files (.h).
- **Make file**—This file contains build instruction. It should be named makefile and consist of the following statement that includes the standard **WDK** make file:

  ```
  !INCLUDE $(NTMAKEENV)\makefile.def
  ```

- **Sources file**—This file contains project-specific information that is used to build the driver, such as the list of source files. The following example shows the content of a basic Sources file. An example of a somewhat more complex Sources file along with an explanation of its element is given in the "Building Featured Toaster," later in this section.

  ```
  TARGETNAME=WdfSimple
  TARGETTYPE=DRIVER
  KMDF_VERSION=1
  SOURCES=WdfSimple.c
  ```

Optional files include

- **Makefile.inc**—Project with custom targets, such as Windows Management Instrumentation-(WMI)-related files, must put the necessary directives in **Makefile.inc**. Do not modify the standard make file.
- **Dirs**—This file is used by projects that have source files in multiple subfolders or to build multiple projects with a single build command.
- **Resource files (.rc)**—These files contain resources such as string tables.
- **Managed object format (MOF) resource files (.mof)**—Drivers that support WMI must have a **MOF resource file (.mof)**.
- **INX** file (**.inx**)—An **INX** file is an architecture-independent **INF** file. When the appropriate instructions are specified, the **Build** utility uses the data in an **INX** file to produce an appropriate **INF** file for the project.

You can use any names that are convenient for most project files and folders, with one important restriction: the names cannot contain spaces or nonANSI characters. However, the build utility assumes by default that the make, **makefile.inc**, **Sources**, and **Dirs** files are named makefile, makefile.inc, sources, and dirs, respectively.

The supporting files must all be created manually. However, you can usually simplify the process by copying the files from an appropriate sample and modifying them to suit the project.

Visual Studio can be used in only a limited way for driver development. In particular, its compiler and debugger are not designed to be used with drivers. However, if you are accustomed to using Visual Studio, you can still use its integrated development environment (IDE) to edit source code and build the driver. Essentially, you reprogram the Visual Studio Build command to bypass the Visual build utility and instead run a command line that launches the **WDK** build utility. You still must manually create the build utility supporting files that were discussed earlier in this section.

KMDF drivers, like WDM drivers, are built in the WDK build environment. KMDF drivers include the header files **Wdf.h** (shipped with KMDF) and **ntddk.h**.

To build a KMDF driver, you must set the /GS flag on the compiler and the **KMDF_VERSION** environment variable in the **Sources** file. Setting **KMDF_VERSION=1** indicates that the driver should be built with the first version of KMDF.

8.2 Build Environment

There are two basic types of build:

- **Checked builds** are similar to the Debug builds that are used in application development. They generate detailed debugging information and enable certain types of debugging-related code such as **ASSERT** macros. **Checked builds** are normally used during the earlier stages of driver development because they are much easier to debug than **free builds**. **Checked builds** are typically somewhat slow.
- **Free builds** are similar to the **Release builds** that are used in application development. They lack the detailed debugging information of a **checked build** and are fully optimized. **Free builds** are more difficult to debug, so they are typically used at the end of the development cycle for final testing and performance tuning.

To simplify the process of setting up the build environment, the **WDK** includes a set of console windows with the correct settings for each build environment/platform/architecture combination.

To open a build environment window:

1. On the taskbar, click **Start**, and then click **All Programs**.
2. Click **Windows Driver Kits**, click the latest **WDK** version, and then click **Build Environments**.
3. Click the appropriate CPU architecture, and then open a **checked** or **free build** environment window for the appropriate Windows version.

The build environment window for a specified version of Windows works for that version and all later versions.

8.3 Building a Project

After you have launched the correct build window, use **cd** to move to the project folder and run the build utility to compile and link the driver. The command syntax is simple:

```
build -a[b[c]...]
```

The build utility assumes by default that the project has a make file that is named **makefile**, a **Sources** file with the list of source files, and so on. There is no need to specify these files explicitly. **a[b[c]...]** represents the build arguments, most of which consists of a single case-sensitive character. The **WDK** has a complete list of flags, but here are some of the commonly used ones:

- **?**—Displays a list of all command-line flags.
- **c**—Deletes all object files.
- **e**—Generates log, error, and warning files.
- **g**—Uses colored text to display warnings, errors, and summaries.
- **Z**—Prevents dependency checking or scanning of source files.

The following example shows a commonly used build command:

```
build -ceZ
```

The build utility produces several output files, including

- **TargetName.sys**—The driver binaries.
- **SourceFileName.obj**—Object files that are produced from the corresponding source files.
- **TargetName.pdb**—The driver's symbol tables.
- **TargetName.inf**—The project's **INF** file. This file is produced by the build utility only if the project uses an **INX** file. Otherwise, you must create the **INF** file separately.

The output normally goes in a subfolder of the project folder. The default output folder name depends on the build environment. For example, the default output folder for a Windows XP x86 free build is named **ProjectFolder\objfre_wxp_x86\i386**.

8.4 Building Featured Toaster

The **Toaster** sample is set of simple software drivers that were created by Microsoft as a learning tool for new driver developers. If you are new to KMDF driver development, it's the first sample you should look at. Not only does **Toaster** provide a simple example of how to write drivers by using the best coding practices, it includes detailed comments that explain every step. **Toaster** is located at **WinDDK\BuildNmber\src\kmdf\toaster**.

Toaster includes a number of drivers, including two function drivers. One is a minimal version that is named **Simple** and the other a full-featured version that is named **Featured**. This section uses the **Featured Toaster** sample as a convenient way to demonstrate the basics of the build, install, and debug process. We will walk through how to create a Windows 7 checked build of **Featured Toaster**. We will discuss some of the supporting files that are mentioned in the previous material. The KMDF version of **Toaster** is essentially a port of the **WDM Toaster** sample. If you are familiar with the **WDM** version, compare it to the KMDF version to see just how much KMDF can simplify driver code.

8.4.1 Makefile and Makefile.inc

The contents of **Makefile** are the same for all driver projects. **Featured Toaster** also includes an optional file, **Makefile.inc**. This file contains some additional make directives that handle two targets that aren't covered by **makefile.def**. The following example shows the contents of **Featured Toaster's makefile.inc** file:

```
_LNG=$(LANGUAGE)
_INX=.
STAMP=$(STAMPINF_PATH) -f $@ -a $(_BUILDARCH) -v 1.0.0.0

$(OBJ_PATH)\$(O)\$(INF_NAME) .inf:  $(_INX)\$(INF_NAME .inx
      copy $(_INX)\$(@B)  .inx $@
   $(STAMP)

mofcomp: $(OBJ_PATH)\$(O)\toaster.bmf

$(OBJ_PATH)\$(O)\toaster.bmf: toaster.mof
   mofcomp -WMI -B:$(OBJ_PATH)\$O\toaster.bmf toaster.mof
   wmimofck -m -h$(OBJ_PATH)\$O\ToasterMof.h  -
w$(OBJ_PATH)\$O\htm $(OBJ_PATH)\$(O)\toaster.bmf
```

The first part of **makefile.inc** uses the project's **INX** file, **wdffeatured.inx**, to produce an architecture-specific **INF** file. The second part of **makefile.inc** produces the **WMI** target file.

8.4.2 The Sources File

The **Sources** file contains most of the project-specific information that the build utility uses to build the project. It consists of a series of directives that assign project-specific values to a set of macros and environment variables. The following example shows the contents of the **Featured Toaster Sources** file, which is a typical **Sources** file for a simple KMDF driver:

```
TARGETNAME=wdffeatured
TARGETTYPE=DRIVER

KMDF_VERSION=1.5

INF_NAME=wdffeatured
MISCFILES=$(OBJ_PATH)\$(O)\$(INF_NAME)  .inf
INCLUDES = $(INCLUDES)  ;..\..\inc;..\shared

NTTARGETFILES=
NTTARGETDILE0=mofcomp

#
# List of source files to compile.
```

```
#
SOURCES=              \
      toaster.rc          \
      toaster.c           \
      power.c          \
      wmi.c

C_DEFINES=

!include $(WDF_ROOT)\project.mk
```

The following list gives brief descriptions of the macros and environment variables in this file. For a complete list, see the WDK documentation.

- **TARGETNAME**—Required. This macro specifies **wdffeatured** as the name to be used for output files such as the project's **.sys** and **.pdb** files.
- **TARGETTYPE**—Required. The build utility can be used to build a variety of binary types. This macro specifies which type of binary is to be built; **DRIVER** indicates a Kernel Mode Driver.
- **KMDF_VERSION**—Required. This environment variable specifies the KMDF version number. This project uses KMDF version 1.5.
- **INF_NAME**—Optional. **INF_NAME** is a custom macro for projects that use **INX** files. It specifies that the **INF** file that is generated from the projects **INX** file is to be named **wdffeatured.inf**.
- **MISCFILES**—Optional. **MISCFILES** is a custom macro for projects that use **INX** files. It specifies where to place the **INF** file that is generated. In this example, the **INF** file is placed in the output folder with the other output files.
- **INNLUDES**—Optional. This macro specifies the location of folders, other than the project folder, that contain header files. These are typically header files that are shared across multiple projects.
- **NTTARGETFILE0**—Optional. This macro is used to specify additional targets and dependencies that are not covered by **makefile.def**. In this case, it is used for **WMI**-related aspects of the build.
- **SOURCES**—Required. This macro lists the project's source files. By default, the files must be on a single line. The backslash (\) is a line-continuation character that allows the files to be listed on separate lines.
- **C_DEFINES**—Required for Unicode builds. Specifies any compile-time switches that must be passed to the C compiler.

8.4.3 The Build

The following procedure shows how to build **Featured Toaster**. For simplicity, assume that the **WDK's** root folder is **C:\WinDDK\6000**.

1. Launch the **Windows 7 Checked Build Environment** console window. It opens in the **c:\WinDDK\6000** folder.
2. Use **cd** to move to the project folder. The project folder is at **C:\WinDDK\6000\src\kmdf\toaster\func\featured**.
3. Build the project by running the following command. This isn't the only way to build the project, but it's a commonly used set of flags.

   ```
   build -ceZ
   ```

The output files go in the **featured\objchk\w7_x86\i386** subfolder.

8.5 Installing a KMDF Driver

Drivers must be installed before they can be used—either by the developer to test and debug the driver or by the end user who wants to use the related device. The procedures for installing a driver are distinctly different from those that are used to install applications. This section discusses how to create a **WDF** installation package and install it on a system.

There is a variety of ways to install drivers on a user's system. One common way is to simply attach the associated hardware to the user's system. The Plug and Play manager detects new hardware and prompts the user to insert a disk that contains the **driver package**. The system then installs the driver. Users can also install drivers manually with the **New Hardware** application in Control Panel.

KMDF driver packages contain at least three files in addition to the driver binaries:

- The KMDF co-installer dynamic-link library (**DLL**).
- An **INF** file.
- A digitally signed catalog (**.cat**) file. This file is not necessary for test installations.

Driver packages can optionally contain files such as icons, property sheet providers, supporting **DLLs**, and so on. This section discusses the relatively simple driver package and test-installation procedure for **Featured Toaster**.

8.5.1 The WDF Co-Installer

A KMDF driver package must include the redistributable **WDF co-install DLL**. Its primary purpose is to install the KMDF run time. The **WDF co-installer** is located under the **WinDDK\BuildNumber\Redis\Wdf** folder. There are six co-installers—a checked and a free build for each supported process architecture (x86, Intel Itanium, and amd64). To install the **WDF co-installer**, add the appropriate **DLL** to the driver package and add the appropriate directives to the **INF** file.

The **WDF co-installer** version number must be greater than or equal to the KMDF version with which the driver is compiled. The version number is embedded in the **DLL**'s name. For example, the co-installer for KMDF version 1.5 is named **WdfCoInstaller01005.dll**. The KMDF run-time version and the KMDF co-installer version that are specified in the project's **INF** must be identical. The build type of the co-installer must match that of the Windows version on which the driver will be installed. You cannot use the checked build of a co-installer to install a driver on a free build of Windows, or vice versa.

8.5.2 The INF

The **INF** is the core of the installation package. It is a text file that contains most of the information that the system uses to install a driver, including

- General information about the device such as the device's manufacturer, installation class, and version number.
- Names and locations of files on the distribution disk and where they should be installed on the user's system.
- Directives for creating or modifying registry entries for the driver or device.
- Installation directives for which drivers are to be installed, which binaries contain the driver, and a list of drivers to be loaded on the device.
- Directives for setting KMDF-specific configuration information.

The **INF** format is much like the earlier Windows **.ini** files. Each line contains a single entry, and there are two basic types of entries:

- **Section**—Each **INF** contains a number of sections, indicated by square brackets—for example, **[Version]**.

- **Directive**—Each section contains one or more directives. A directive is a key-value pair and is used to specify various types of installation-related data. For example, the **Class=Mouse** directive in the **Version** section specifies the mouse device class.

8.5.3 INFs for KMDF Drivers

Most of the contents of an **INF** for a KMDF driver are similar to those that are used for **WDM** drivers and aren't discussed here. For further information, see the **WDK** document or examine the **INF** for **Featured Toaster**, **wdffeatured.inf**. The major difference is that **INF**s for KMDF drivers must contain several additional sections that are devoted to the KMDF co-installer. These sections instruct the system to run the co-installer and provide it with necessary data. The co-installer unpacks and installs a number of files that KMDF drivers require, including the KMDF run-time library.

8.5.4 wdffeatured.inf

The following sample shows the **WDF co-installer** sections from the **Featured Toaster** sample's **INF** file, **wdffeatured.inf**. It was produced from an **INX** file by the build described earlier.

```
[DestinationDirs]
ToasterClasInstallerCopyFiles = 11

[Toaster_Device.NT.CoInstallers]
AddReg=Toaster_Device_CoInstaller_AddReg
CopyFiles=Toaster_Device_CoInstaller_CopyFiles

[Toaster_Device_CoInstaller_AddReg]
HKR, ,CoInstallers32, 0x00010000,
"WdfCoinstaller01000.dll,WdfCoInstaller"

[Toaster_Device_CoInstaller_CopyFiles]
WdfCoinstaller01000.dll

[SourceDisksFiles]
WdfCoinstaller01000.dll=1

[Toaster_Device.NT.Wdf]
KmdfService = wdffeatured, wdffeatured_wdfsect
```

```
[wdffeatured_wdfsect]
KmdfLibraryVersion = 1.1
```

To modify this code for your driver, replace the text that is specific to **Featured Toaster** with custom text of your choosing. The following example is a generic version of the co-installer section for a driver that is named **MyDevice**:

```
[DestinationDirs]
MyDeviceClassInstallerCopyFiles = 11

[MyDevice.NT.CoInstallers]
AddReg=MyDevice_CoInstaller_AddReg
CopyFiles= MyDeviceClassInstallerCopyFiles

[MyDevice_CoInstaller_AddReg]
HKR, ,CoInstallers32,0x00010000,
"WdfCoinstaller01000.dll,WdfCoInstaller"

[MyDevice_CoInstaller_CopyFiles]
wdfCoinstller01000.dll

[SourceDisksFiles]
wdfCoinstaller01000.dll=1  ;

[MyDevice.NT.Wdf]
KmdfService = MyDevice, MyDevice_wdfsect
[MyDevice_wdfsect]
KmdfLibraryVersion = 1.0
```

8.6 Catalog Files and Digital Signature

Because Kernel Mode Drivers have essentially unrestricted access to the system, they should be digitally signed. Digitally signing the package simplifies the installation process, but it also provides customers with two very important additional benefits:

- Customers can use the signature to identify the origin of the package.
- Customers can use the signature to verify that the contents of the package have not been tampered with since it was signed. For example, this assures them that the driver has not been modified into a root kit or infected with a virus.

With recent versions of Windows, unsigned drivers can be installed only by an administrator, and even administrators receive a warning dialog box that requires them to explicitly approve the installation.

A signed catalog file (**.cat**) contains the digital signature for the entire driver package. The signing process ties the catalog file to a specific driver package. If anyone subsequently modifies any member of the package by even a single byte, it invalidates the signature. If you modify a driver package, it must have a new signed catalog file.

There are two ways to obtain a signed catalog file for a driver package:

- Obtain a Windows logo. Drivers that pass the Windows Hardware Quality Lab (WHQL) testing and receive a Windows logo also receive a catalog file for the driver package, signed with the WHQL certificate.
- Create your own signed catalog file. You can obtain a digital certificate from a certificate authority (CA). The **WDK** provides tools to create a catalog file and sign it with the certificate.

For testing purposes, you can create a test certificate and install it in the trusted publishers' certificates store on the test computer. Sign the test driver packages with the test certificate and the driver will install without warning messages.

The **CatalogFile** entry in the **INF** file's **Version** section specifies a package's catalog file. The following example is from the **Featured Toaster** sample's **INF** file and declares **KmdfSamples.cat** as the package's catalog file:

```
[Version]
Signature="$WINDOWS NT$"
Class=TOASTER
ClassGuid={B85B7C50-6A01-11d2-B841-00C04FAD5171}
Provider=%MSFT%
DriverVer=02/22/2006, 1.0.0.0
CatalogFile=KmdfSamples.cat
```

8.7 Installing Featured Toaster

Kernel Mode Drivers under development are normally installed on a separate test computer that is used specifically for testing and debugging drivers. If you are new to drivers, there are two primary reasons for this practice:

- Kernel Mode Drivers have essentially unrestricted access to the system. This means that a misbehaving driver can corrupt system memory and possibly the contents of the hard disk. Drivers under development invariably have bugs, and it is better to have any related damage happen to a stripped-down test computer that can be easily reformatted.
- Debugging Kernel Mode Drivers normally requires two computers: one to host the driver being debugged and one to host the debugging software. One important reason for this arrangement is that driver bugs often hang or crash the system. Hosting the debugger on a separate system protects it from crashing along with the target computer and allows you to immediately analyze the problem.

This section describes how to install **Featured Toaster** on a test computer. The driver is installed on a **root-enumerated** physical device object, which is the simplest approach. More commonly, drivers are installed on a **bus-enumerated** physical device object. The **Toaster** sample also includes a bus driver that can be used for this type of installation.

Remember that **Featured Toaster** is a software driver, not a device driver. This means that there is nothing for the Plug and Play manager to detect, so the driver must be installed manually. Because the test driver is unsigned, installing it requires administrator rights and an extra step. For a more streamlined process, install a test certificate on the test computer and use the certificate to sign the package. The steps involved in installing the **Featured Toaster** are as follows:

1. Copy the driver binary **(WdfFeatured.sys)** and **INF** file **(WdfFeatured.inf)** to installable media such as a USB drive.
2. Copy the **WDF co-installer** to the same media.
3. Put the media on the test computer, start the Control Panel **Add Hardware** wizard, and go to page 2.
4. Page 2: Click Yes, I Have Already Added the Hardware.
5. Page 3: Select Add a New Hardware Device, from the bottom of the list.
6. Page 4: Click Install the Hardware That I Manually Select From a List.

7. Page 5: Select Show All Devices from the top of the list. It may take awhile for page 6 to appear.
8. Page 6: Click Have Disk, which opens the Install From a Disk dialog box.
9. Enter the drive letter for the media that contains the driver package and click OK to return to the wizard.
10. Page 8: Select the **Featured Toaster** driver from the list and click Next on this page and the following page. The system then loads the driver.
11. Page 10: Click Finish to complete the process.

Device Manager is the simplest way to uninstall the driver. On Windows 7, you can also use System Restore to restore the system to the state it had before the driver installation—that uninstalls the driver, along with any other system changes that took place in the interim.

8.8 Testing a KMDF Driver

Testing drivers is a large and complicated subject. This section touches on only a few KMDF-specific issues.

There are two basic approaches to testing:

- Static testing by using tools that analyze the source code for errors without actually executing it.
- Dynamic testing that puts an installed driver through its paces in hopes of activating a bug and causing the driver to fail in some way.

Some related techniques, such as tracing tools, record the actions of a driver. This section only provides a brief introduction to the testing tools that WDF provides for KMDF drivers.

8.8.1 PREfast

PREfast is a static source code analysis tool that detects certain classes of errors not easily found by a compiler. PREfast steps through all possible execution paths in each function and evaluates each path for problems by simulating execution. PREfast does not actually execute code and cannot find all possible errors. However, it can find errors that the compiler might not catch and that can be difficult to find during debugging.

PREfast is a general-purpose tool that can be used with any type of project. WDF includes a customized version of PREfast that checks for driver-specific issues such as the correct interrupt request level (IRQL), use of preferred driver routines, and misuse of driver routines. It also aggressively checks for memory and resource leaks.

PREfast is run in conjunction with a build. The following is a simple example of how to run a PREfast build. For the purposes of illustration, the command uses a typical set of build flags, but any build flags can be used with PREfast. The second line opens the PREfast viewer to display the error log.

```
prefast       build -ceZ
prefast       view
```

8.8.2 Static Driver Verifier

Static Driver Verifier (SDV) is a static compile-time unit-testing tool that symbolically executes the source code. SDV does deeper testing than PREfast and creates what is in effect a hostile environment for the driver. It systematically tests all code paths by looking for violations of usage rules. The symbolic execution makes very few assumptions about the state of the operating system or the initial state of the driver, so SDV can create scenarios that are difficult to handle with traditional testing.

The set of rules that are packaged with SDV define how device drivers should use the DDI. The categories of rules tested include the following.

Category	Tests
IRP	Functions that use I/O request packets
IRQL	Functions that use interrupt request levels
PnP	Functions that use Plug and Play
PM	Functions that use power management
WMI	Functions that use Windows Management Instrumentation
Sync	Functions that use synchronization, including spin locks, semaphores, timers, mutexes, and other methods of access control
Other	Functions that are not fully described by any of the other categories

8.8.3 KMDF Log

KMDF includes an internal trace logger that is based on the Windows software trace preprocessor (WPP). It tracks the progress of I/O request packets (IRPs) through the framework and the corresponding **WDFREQUEST** objects through the driver. The KMDF log maintains a record of recent race events—currently, approximately the last 100—for each driver instance. Each KMDF driver has its own log.

You can use WDF debugger extensions to view and save the KMDF log during interactive debugging. The typical saved log file is small (10 to 20 KB) and written in a binary format. You can also make logs available as part of a small-memory dump for inspection after a crash.

8.8.4 KMDF Verifier

KMDF **Verifier** operates on an installed and running driver. It complements **Driver Verifier** and supports a number of WDF-specific features. In addition, if the target driver is not loaded, KMDF **Verifier** can be turned on without rebooting the system. In general, you should run both **Driver Verifier** and KMDF **Verifier** during development.

KMDF **Verifier** provides extensive tracing messages that supply detailed information about activities within the framework. It tracks references to each WDF object and builds a trace that can be sent to the debugger. In particular, KMDF **Verifier**

- Checks lock acquisition and hierarchies.
- Ensures that calls to the framework occur at the correct IRQL.
- Verifies correct I/O cancellation and queue usage.
- Ensures that the driver and framework follow the documented contracts.

KMDF **Verifier** can also simulate low-memory and out-of-memory conditions. It tests a driver's response to these situations to determine whether the driver responds properly without crashing, hanging, or failing to unload.

8.8.5 Debugging a KMDF Driver

Debuggers are an essential development tool; programs under development always have bugs, especially in the early stages. Debuggers can also be used as learning tools, to step through sample code and understand in detail how it functions.

Debugging is normally done at run time to determine why a driver is failing. The exception to this rule is that kernel debuggers can also be used to analyze crash-dump files. If you are new to driver development, you will find kernel debugging a bit different from application debugging. One immediately noticeable difference is that kernel debugging requires three hardware components:

- A host computer running WinDbg. This is typically the computer that is used to develop and build the driver.
- A test computer running an appropriate build of Windows with the driver installed and kernel debugging enabled. Debugging is typically done with a checked build of the driver because checked builds are much easier to debug. Test computers also often run a checked build of Windows.
- A way for the two computers to communicate. Historically, this was handled by connecting serial ports on the host and test computers with a null-modem cable. An alternative is to use USB or IEEE 1394 cables.

The kernel debugging tools are available as a separate package from WHDC that includes the debugging tools, documentation, and some related files. The procedures for setting up systems for kernel debugging are covered in detail in the debugging documentation.

The debugging package includes two kernel debuggers, WinDbg and KD. They have essentially the same capabilities, but WinDbg has a graphical user interface (GUI) that many developers find convenient. The examples we use are from WinDbg. We will cover a walkthrough of a simple debugging session with **Featured Toaster** that demonstrates the basics of how to use WinDbg with a KMDF driver.

WinDbg is a debugger, not an IDE like Visual Studio. It comes into play only after you have successfully built the driver and installed it on a test machine. There are two basic ways to use WinDbg:

- Kernel debugging—In this mode, WinDbg is connected to an active test machine and can interact with a running driver.
- Crash dump analysis—If the system crashes, you can use WinDbg to analyze the crash dump data to try to determine the cause.

When you launch WinDbg, you must first point it to the driver's source and symbol files. To start a kernel debugging session, on the **File** menu, click **Kernel Debug**. You won't be able to do much until the system breaks into the debugger. This essentially stops the test computer and turns its operation over to WinDbg. The following are common ways to cause a test system to break into the debugger:

- You instruct WinDbg to force a break. This can be done from the UI, on the **Debug** menu by clicking **Break**, or by clicking the corresponding toolbar button. You can also run the break command.
- You use WinDbg to dynamically insert breakpoints into the running driver. This approach is quite flexible because it allows breakpoints to be inserted, disabled, enabled, or removed during the debugging session.
- You insert DbgBreakPoint statements in the driver's source code. This approach is simpler but less flexible because the driver must be recompiled and reinstalled to change a breakpoint.
- The driver bug checks and crashes the test computer. At this point, you can use WinDbg to examine crash dump data, but the computer must be rebooted before it can run again. You can force a system crash by running the **.crash** command.

8.8.6 Kernel Debugging

With the first two cases in the previous discussion, after the driver breaks into the debugger, you can do most of the usual debugging procedure: examine variables, step through lines of code, examine the call stack, and so on.

Much of the interaction with WinDbg is through the command-line interface. There are two basic types of commands:

- **Debugger commands** are native to the debugger and are used to obtain basic information. Commands are typically one or two letter strings, often followed by one or more arguments. For example, **k** and related commands display a thread's stack frame and some related information. When you are finished, use the **g** command to break out of the debugger and return the driver and test computer to normal operation. Some simple commands have corresponding menu items or toolbar buttons, but many can be run from the command line.

- **Debugger extensions** extend the basic set of debugger commands. A number of them are included with the debugger package and are launched from the command window in much the same way as debugger commands. The first character of a debugger extension is always an exclamation point (!), to distinguish it from a debugger command. For example, a particularly useful debugger extension is **!analyze**, which is used to analyze crash dumps. In addition to the debugger extensions that are included with the debugger package, it is also possible to write custom debugger extensions.

The debugging **Help** file includes a complete reference for debugging commands, standard debugger extensions, and the API that is used to create custom extensions. In the next section, we will discuss some debugger extensions that were created specifically for KMDF drivers.

If a driver bug causes a system crash, the computer must be rebooted before it can run again. However, if WinDbg is running and connected when the test computer crashes, the system breaks into the debugger and you can analyze the crash dump immediately. You can also configure your test computer to attempt to create a crash dump file when it crashes. If the file is successfully created, you can load it into WinDbg and analyze the crash after the fact. WinDbg doesn't have to be connected to the test computer for this purpose.

8.8.7 KMDF Driver Features

Debugging a KMDF driver is similar in many ways to debugging any Kernel Mode Driver. However, some debugging features are specific to KMDF.

8.8.7.1 Registry Settings

A number of the WDF debugging features must be enabled by setting registry values for the driver's **Parameters\Wdf** subkey. The driver key itself is named for the driver and located under **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services**.

Table 8.1 summarizes the values that can be added to the **Wdf** subkey. The features are disabled by default. To enable most of these features, create the associated value and set it to a nonzero number. To enable handle tracking, set **TrackHandles** to a **MULTI_SZ** string that contains the names of the objects that you want to track. The settings do not take effect until the next time that the driver is loaded. The simplest way to reload a driver is to use **Device Manager** to disable and then reenable the driver. Table 8.1 shows some of the settings.

Table 8.1 Registry Values

Category	Type	Tests
VerifierOn	REG_DWORD	Set to a nonzero value to enable the KMDF verifier.
VerifyOn		Set to a nonzero value to enable the WDFVERIFY macro. If VerifierOn is set, WDFVERIFY is automatically enabled.
DbgBreakOnError	REG_DWORD	Set to a nonzero value to instruct the framework to break into the debugger when a driver calls WdfVerifierDbgBreakPoint.
VerboseOn	REG_DWORD	Set to a nonzero value to capture verbose information in the KMDF logger.
LogPages	REG_DWORD	Set to a value from 1 to 10 to specify the number of memory pages that the framework assigns to its logger. The default value is 1.
VerifierAllocateFailCount	REG_DWORD	Set to a nonzero value to test low-memory conditions. When VerifierAllocateFailCount is set to n, the framework fails every attempt to allocate memory for the driver's objects after the nth allocation. This value works only if VerifierOn is also set.
TrackHandles	MULTI_SZ	Set to a MULTI_SZ string that contains the names of one or more object types to track handle references to those types. This feature can help find memory leaks that are caused by unreleased references. To track all objects types, set TrackHandles to "*".

Category	Type	Tests
ForceLogsInMiniDump	REG_DWORD	Set to a nonzero value to include the KMDF log in a small memory dump file if the system crashes.

8.8.7.2 Symbols

You must explicitly provide WinDbg with paths to all the relevant symbols files. For KMDF drivers, this normally includes the symbols for the driver, Windows, and the KMDF run time. The symbols file is named **Wdf01000.pdb** and is located under **%WDF_ROOT%\Symbols\Build-Environment\wdf\sys**. There are six Build Environment folders, one for each of the standard build types and architectures. For most debugging, you should use the checked build for the appropriate architecture.

8.9 Debugging Macros and Routines

KMDF drivers can include any of the standard debugging macros and routines, such as **ASSERT** or **DbgPrintEx**. Several routines and macros are also specific to WDF as summarized in Table 8.2.

Table 8.2 Debugging Macros

Category	Tests
WdfVerifierDbgBreakPoint	This routine breaks into the debugger if the DbgBreakOnError value is set in the registry.
WDFVERIFY	If the VerifyOn value is set in the registry, this macro tests a logical expression and breaks into the kernel debugger if the expression evaluates to FALSE. Unlike ASSERT, this macro is included in both checked and free builds.
VERIFY_IS_IRQL_PASSIVE_LEVEL	If the VerifyOn value is set in the register, this macro breaks into the kernel debugger if the driver is not executing at IRQL=PASSIVE_LEVEL.

8.10 WDF Debugger Extension Commands

KMDF includes a set of debugger extension commands that can be invoked in the debugger's command window to obtain a variety of WDF-related data. The output often includes command strings that can be pasted into the command line to retrieve additional related information. Table 8.3 shows some of the more commonly used commands.

Table 8.3 Debugger Extension Commands

Category	Tests
!wdfhelp	Displays the list of debugger extensions.
!wdfcrashdump	Displays a crash dump that includes the framework's log information.
!wdfdevice	Displays information that is associated with a WDFDEVICE-typed handle.
!wdfdevicequeues	Displays information about all the queue objects that belong to a specified device.
!wdfdriverinfo	Displays information about a framework-based driver, such as its run-time version and hierarchy of object handles.
!wdfhandle	Displays information about a specified KMDF handle.
!wdfIotarget	Displays information about a WDFIOTARGET-typed object handle.
!wdfdr	Displays all loaded WDF drivers.
!wdfIodump	Displays the framework's log information.
!wdfqueue	Displays information about a WDFQUEUE-typed object handle.
!wdfrequest	Displays information about a WDFREQUEST-typed object handle.

When KMDF **Verifier** is enabled, several of the WDF debugger extension commands provide more information than is available otherwise. For example, **!wdfdriverinfo** reports leaked handles.

The code for the WDF debugger extensions is contained in a DLL that is named **wdfkd.dll**. The DLL is included with WDK debugging tools and is stored in **Program Files\Debugging Tools for Windows\winext**.

Check it against the most recent version of the DLL, which is included with the latest KMDF distribution. **Wdfkd.dll** is stored under **WinDDK\BuildNumber\bin\**. There are actually three versions of **wdfkd.dll**, one for each supported architecture (amd64, Intel Itanium, or x86). If necessary, overwrite the version that came with the debugger with the most recent version.

8.11 Using WPP Tracing with a KMDF Driver

WPP tracing works in essentially the same way with a KMDF driver as it does with a WDM driver. However, KMDF provides hundreds of framework-specific tracing messages. To enable KMDF support for WPP, include the following **RUN_WPP** directive in the project's **Sources** file:

```
RUN_WPP = $(SOURCES)                        \
          -km                               \
          -func:TraceEvents(LEVEL,FLAGS,MSG,...)    \
          -gen:{km-WdfDefault.tpl}*.tmh
```

8.12 Using WinDbg with Featured Toaster

Featured Toaster doesn't have any known bugs, but WinDbg is still a useful tool to walk through the source and see how the driver works. The walkthrough also demonstrates the basics of how to use WinDbg.

The test computer should be running Windows with kernel debugging enabled and be connected to the host computer. For convenience, this section assumes that the COM ports are connected with a null-modem cable.

The simplest way to use the debugger is to run a test application that accesses the driver and hits a breakpoint, causing the driver to break into the debugger. KMDF doesn't include a test application for **Toaster**, but the WDM test application, **Toast.exe**, works just as well with the KMDF version of the driver.

The source code for **Toast.exe** is located at **WinDDK\BuildNumber\src\general\toaster\exe\toast**. Use the same console window and build commands to build **Toast.exe** as for **Featured Toaster**. Unlike most projects, the output folder is not a subfolder of the project folder. Copy **Toast.exe** from the output folder **WinDDK\BuildNumber\src\general\toaster\disk\chk_wxp_x86\i386** to a convenient folder on the test computer.

To get WinDbg ready to debug **Featured Toaster:**

1. Launch WinDbg.
2. On the **File** menu, click **Symbol File Path,** which causes the **Symbols Search Path** dialog box to appear. It should have paths for the **Featured Toaster**, Windows, KMDF run time, and **wdfkd.dll** symbols files.
3. Select the **Reload** check box, which forces WinDbg to load the current symbols, and close the dialog box.
4. On the **File** menu, click **Source File Path**, and add the path to the **Featured Toaster's** source files.
5. On the **File** menu, click **Open Source File**. Open **Featured Toaster's Toaster.c** file.
6. On the **File** menu, click **Kernel Debug**. This puts WinDbg into kernel debugging mode and establishes the connection with the test computer.
7. Enter the appropriate baud rate and COM port in the **Kernel Debugging** dialog box, and click **OK** to start the debugging session.
8. On the **Debug** menu, click **Break**, which forces a break and allows you to run debugging commands.
9. Use the **bp** command as follows to set a breakpoint in **Featured Toaster's ToasterEvtIoRead** routine.

   ```
   bp ToasterEvtIoRead
   ```

To start debugging:

1. Go to the test computer, launch a command window, and run **Toast.exe**. Type any character other than "q" to cause **Toast.exe** to send a read request to the driver. **Toast.exe** then calls **Featured Toaster's ToasterEvtIoRead** routine. When the driver hits the breakpoint, it breaks into the debugger.
2. On the **Debug** menu, select the **Source Mode** check box if it isn't already selected. This mode allows you to step through the source code. Notice that the corresponding assembler appears in the **Command** window. Even if you never write a line of assembler, it's still useful to know something about it for debugging purposes.
3. Set the cursor on a line of source code. On the **Debug** menu, click **Run to Cursor**. The selected line should be highlighted in blue.
4. On the **Debug** menu, click **Step Over** to execute the next line. There's also a **Toolbar** button for this purpose, located underneath the **Window** menu.

To get detailed information, call one of the debugger extensions. One useful WDF debugger extension is **!wdfdriverinfo**, which returns general information about the driver. It takes the name of the driver as a required argument plus a flag that controls exactly what data is returned. 0xF0 returns essentially everything. The following example shows the output from **!wdfdriverinfo** for **Featured Toaster**:

```
kd> !wdfdriverinfo wdfeatured 0xf0

Default driver image name:      wdffeatured
WDF runtime image name:       Wdf01000
  FxDriverGlobals    0x829a3c80
  WdfBindInfo        0xf7991b8c
     Version         v1.0 build(1234)

Driver Handles:
WDFDRIVER    0x7d6494f8    dt    FxDriver    0x829b6b00
   WDFDEVICE     0x7d4e1588    dt  FxDevice    0x82b1ea70
     Context   82b1ec30
Cleanup   f7992dc0
   WDF    INTERNAL    dt   FxDefaultIrpHandler   0x82aca158
   WDF    INTERNAL    dt   FxPkgGeneral    0x829e8160
   WDF    INTERNAL    dt   FxWmiIrpHandler   0x82aa8cb0
   WDF    INTERNAL    dt   FxPkgIo    0x82b1cef0
      WDFQUEUE   0x7d4e1250  dt   FxIoQueue    0x82b1eda8
      WDFQUEUE   0x7d552720  dt   FxIoQueue    0x82aad8d8
   WDF    INTERNAL    dt   FxPkgFd0    0x82aa9d50
   WDFCMRESLIST    0x7d4e6340   dt   FxCmResList    0x82b19cb8
   WDFCMRESLIST    0x7d4e1e10   dt   FxCmResList    0x82b1e1e8
   WDFCHILDLIST    0x7d639ba8   dt   FxChildList    0x829c6450
   WDFIOTARGET     0x7d6462d8   dt   FxIoTarget     0x829b9d20
   WDF    INTERNAL           dt   FxWmiProvider    0x829b6698
      WDF   INTERNAL    dt   FxWmiInstanceExternal   0x82b1e098
   WDFWMIPROVIDER  0x7d644088   dt   FxWmiProvider  0x829bbf70
      WDFWMIINSTANCE   0x7d560080   dt   FxWmiInstanceExternal
0x82a9ff78   Context   82a9ffe8
   WDFWMIPROVIDER  0x7d6436d0   dt   FxWmiProvider   0x829c928
      WDFWMIINSTANCE   0x7d550d78   dt   FxWmiInstanceExternal
0x82aaf280
   WDFWMIPROVIDER  0x7d6432a8   dt   FxWmiProvider   0x829bcd50
      WDFWMIINSTANCE   0x7d5295e0   dt   FxWmiInstanceExternal
0x82ad6a18   Context   82ad6a88
   WDFFILEOBJECT   0x7d4edc18   dt   FxFileObject   0x82b123e0
```

Many debugger commands and extensions require a handle to an object. For example, **!wdfrequest** takes a **WDFREQUEST** object handle and returns information about the object. To get such a handle, on the **View** menu, click **Locals**. Assuming that the debugger is still in the **ToasterEvtIoRead routine**, the associated **WDFREQUEST** object is name **Request**. The handle appears in the corresponding **Value** field in the **Locals** window.

8.13 Versioning and Dynamic Binding

When Windows loads a KMDF driver, the driver is dynamically bound to the KMDF run-time library (**WdfMM000.sys**). Multiple drivers can share the same run-time library (DLL) image, and the run-time libraries for two major versions can co-exist side-by-side.

When you build a KMDF driver, you link it with **WdfDriverEntry.lib**. This library contains information about the KMDF version in a static data structure that becomes part of the driver binary. The internal **FxDriverEntry** function in **WdfDriverEntry.lib** wraps the driver's **DriverEntry** routine, so that when the driver is loaded, **FxDriverEntry** becomes the driver's entry point. At load time, the following occurs:

- **FxDriverEntry** calls the internal function **WdfVersionBind** (defined in **wdfldr.sys**) and passes the version number of the KMDF run-time library with which to bind.
- The loader determines whether the specified major version of the framework library is already loaded. If not, it starts the service that represents the framework library and loads the library and the driver. If so, it adds the driver as a client of the service and returns the relevant information to the **FxDriverEntry** function. If the driver requires a newer version of the run-time library than the one already loaded, the loader fails and logs the failed attempt in the system event log.
- **FxDriverEntry** calls the driver's **DriverEntry** function, which in turn calls back to KMDF to create the KMDF driver object.

Although two major versions of KMDF can run side-by-side simultaneously, two minor versions of the same major version cannot. At installation, a more recent minor version of the KMDF run-time library overwrites an existing, older minor version. If the older version is already loaded when a user attempts to install a driver with a newer version, the user must reboot the system.

For a boot driver, the loading scenario is different because the KMDF run-time library must be loaded before the driver. At installation, the co-installer reads the **INF** (or the registry) to determine whether the driver is a boot driver. If so, the co-installer both changes the start type of the KMDF service so that the Windows loader starts it at boot time and sets its load order so that it is loaded before the client driver.

This page intentionally left blank

CHAPTER 9

Programming Drivers for the Kernel Mode Driver Framework

The Windows Driver Framework (WDF) release includes several sample Kernel Mode Drivers. You can use sample drivers as a basis for your own drivers and refer to them for examples of specific implementation techniques. The KMDF sample drivers are installed in the **Src\Kmdf** subdirectory of the WDF installation directory.

This chapter lists and discusses the samples in three ways: by name, by device function, and by the features that they demonstrate. We will show various listings of the various sample drivers and discuss the listing. Table 9.1 lists the samples by name.

Table 9.1 KMDF Samples by Name

Sample Name	Description
1394	Virtual (1394vdev.sys) and physical (1394diag.sys) diagnostic drivers for IEEE 1394 devices that interface with the upper edge of the 1394 stack.
AMCC5933	Sample driver for devices based on or similar to the AMCC5933 chip for PCI or ISA.
Echo	Demonstration driver that does not control any hardware. It uses a serial I/O queue to serialize read and write requests that are targeted at the device and shows how to handle request cancellation.
FakeModem	Driver for controllerless modem (soft modem).

(continues)

Table 9.1 KMDF Samples by Name *(continued)*

Sample Name	Description
Firefly	Filter driver for a human interface device (HID).
Kbfiltr	Filter driver for a keyboard. It exposes a raw physical device object (PDO) for communication with user-mode application.
NdisEdge	Driver that exposes an NDIS miniport interface at its upper edge and interacts with other drivers such as USB, IEEE1394, and serial at its lower edge.
NdisProt	Driver that exposes a WDF interface at its upper edge and an NDIS interface at its lower edge.
Nonpnp	Legacy, NT 4.0-style driver that does not support Plug and Play or interact with hardware. It handles four different device I/O control (IOCTL) requests and shows how to handle METHOD_NEITHER I/O. It provides kernel-level services to a user application, which dynamically loads and unloads it.
OsrUsbFx2	Driver for OSR USB-FX2 Learning kit. It shows how to perform bulk and isochronous data transfers to a generic USB device and how to write a bus driver.
Pcidrv	Function driver for Intel 82557/82558-based PCI Ethernet Adapter (10/100) and Intel compatibles.
Pix9x5x	Sample function driver for devices based on the PLx PCI19056RDK-Lite Adapter, a PCI/DMA device.
Ramdisk	RAM disk driver that shows how to create a virtual disk in memory.
Serial	WDF version of the in-box serial driver.
Toaster	Function, filter, and bus drivers for a hypothetical toaster device.

If none of the sample drivers supports your specific device type, you might be able to find a sample that supports a device that has similar characteristics or is used in a similar way. Table 9.2 lists the device characteristics and usage models that the KMDF samples support.

Table 9.2 KMDF Samples by Device Usage Model

Device Usage Model	Device or Driver Type	KMDF Sample
Hardware that supports only port-mapped I/O Driver that handles I/O operations serially and reads or writes one port at a time Driver that polls read operation at regular intervals from either a deferred procedure call (DPC) or a dedicated kernel thread	Parallel port, legacy joystick port, and ISA devices	AMCC5933 ISA sample (S5933DK1)
Hardware that supports port-mapped I/O and interrupts to notify the driver about input data and other asynchronous events	Serial port, parallel port, IDE controller, and PS/2 controller	Serial
Same as previous, but also supports memory-mapped I/O	Typical PCI and EISA devices for data acquisition that use direct memory access (DMA)	PLX9x5x
Same as previous, but also supports bus master DMA channels to read and write	Network adapters and similar PCI drivers	PCIDRV
Hardware that has more than one function or emulates more than one device	Multifunction PCI devices that do not confirm to the PCI specification, multiport serial cards, and multiport network cards	Toaster bus driver
Driver that supports a virtual bus	Serial cards, and multiport network cards	

(continues)

Table 9.2 KMDF Samples by Device Usage Model *(continued)*

Device Usage Model	Device or Driver Type	KMDF Sample
Filter driver that modifies I/O requests and provides an interface for applications to directly control the filter	Keyboard and mouse filters, storage class filter drivers, and serial devices	Toaster filter driver
Filter driver that modifies the hardware resources		
Driver that interacts with an Unrelated device stack to perform I/O	NDIS protocol drivers, Asyncmac, transport driver interface (TDI) client drivers, and Ftdisk or volsnap	NdisEdge and NdisProt
Driver that supports a USB client	Any USB device	Osrusbfx2
Software-only drivers or drivers that are not part of any Plug and Play stack	No device or legacy devices	Nonpnp
Legacy NT 4.0-style drivers that do not support Plug and Play		
Driver that must run in the context of the user application so that it can handle METHOD_IOCTs or map memory into user address space	Video capture devices, audio cards, and high-speed data acquisition devices	Nonpnp
Drivers that registers in-process context callback to handle METHOD_NEITHER I/O requests		

You can also refer to the samples to find out how to use specific KMDF features. Table 9.3 lists the samples that support each fundamental feature.

Table 9.3 KMDF Feature Support in Samples

KMDF Feature	Sample
Buffered I/O	Ndisprot
Child device enumeration	OsrUsbFx2, Toaster bus
Collection	PCIDRV, Toaster bus
Direct I/O	Ramdisk
DMA	AMCC5933, PCIDRV, PLX9x5x
DPC	PCIDRV, PLX9x5x
Event tracing	AMCC5933, NonPnp, OxrUsbFx2, PCIDRV
Functional device object (FDO) interface	All samples
File object	NonPnP
Filter driver	Firefly, Kbfiltr
I/O request cancellation	Echo
I/O requests and I/O queues (serial/parallel/manual)	All samples
I/O target objects	1394, Firefly, NdisEdge, OsrUsbFx2
Idle detection	PCIDRV, serial
In-process callback to handle events in caller's thread context	NonPnP
Interrupt handling	PLX9x5x, Serial
Memory pool	All samples
METHOD_NEITHER I/O	NonPnP
Non-Plug and Play, NT 4.0 style device objects (also called control device objects)	Ndisprot
Physical Device object (PDO) interface	Kbfiltr, OsrUsbFx2/EnumSwitches, Toaster bus
Plug and Play device interface	AMCC5933, PCIDRV, PIX9x5x, Toaster function
Plug and Play hardware resources	PCIDRV, PLX9x5x, AMCC5933
Plug and Play query interfaces	Toaster bus
Power policy owner	PCIDRV

(continues)

Table 9.3 KMDF Feature Support in Samples *(continued)*

KMDF Feature	Sample
Preprocessing callbacks for IRP_MJ_FLUSH_BUFFERS, IRP_MJ_QUERY_INFORMATION, and IRP_MJ_SET_INFORMATION	Serial
Raw PDO	Kbfiltr, OsrUsbFx2
Registry	Fakemodem, PCIDRV, Ramdisk, Serial, Toaster bus
Self-managed I/O	Echo, PCIDRV
Symbolic links to device names	Fakemodem, Ramdisk
Synchronization scope	Echo
Timer objects	Echo
USB device support	OsrUsbFx2
Wake signal support	OsrUsbFx2, PCIDRV, Serial
Windows management instrumentation (WMI)	Firefly, PCIDRV, Serial, Toaster function
Work items	AMCC5933, PCIDRV

9.1 Differences between KMDF and WDM Samples

The KMDF samples are based on the similarly named Windows Driver Model (WDM) samples that are provided in the Windows Driver Kit (WDK). With a few exceptions, the corresponding drivers support similar features. If you are experienced with WDM, you might find useful a comparison of the KMDF and WDM samples.

The primary difference between the samples is that the KMDF samples are much shorter and less complex. The reason is that KMDF implements most of the details of WMD, so that you can avoid writing many lines of code that perform common tasks and implement common features required in all drivers. Instead, you define callbacks for the conditions and events that your driver must handle.

For example, KMDF driver, like WDM drivers, support Plug and Play and power management for their devices. WDM drivers typically include thousands of lines of code to ensure that they handle every possible state and minor I/O request packet (IRP) code; drivers often require code to handle IRPs that they don't even support. A KMDF driver, however, includes code to handle only those features and requests that its device supports. All other Plug and Play and power requests are handled by WDF in a default manner. These defaults make possible the incremental development of KMDF drivers. You can implement a skeletal set of features, test the implementation, and then incrementally add code to support additional features or perform more complicated tasks.

In a practical sense, the most significant difference between KMDF and WDM drivers is the number and complexity of the required functions. For an example of this difference, compare the simplest form of the WDM Toaster sample (src/general/toaster/func/Incomplete1 in the WDK installation directory) with the simplest KMDF Toaster sample (src/kmdf/toaster/func/simple in the WDF installation directory).

Both drivers support basic Plug and Play, power, and WMI requests. The KMDF sample also includes stub functions to handle read, write, and IOCTL requests. However, the KMDF sample contains many fewer lines of code than the WDM sample. Table 9.4 compares the functions in these two drivers.

Table 9.4 Comparison of a KMDF Simple Toaster and WDM incomplete1 Toaster Samples

WDM Function	Equivalent KMDF Function	Comments
DriverEntry	DriverEntry	Function has same prototype. KMDF driver creates WDF driver object.
ToasterAddDevice	ToasterEvtDeviceAdd	KMDF driver creates a default queue for I/O requests.
ToasterUnload	None	KMDF driver uses defaults.
ToasterDispatchPnP	None	KMDF driver uses defaults.
ToasterSendIrp-Synchronously	None	WDM driver passes IRPs down the stack to be completed synchronously. KMDF driver uses defaults.

(continues)

Table 9.4 Comparison of a KMDF Simple Toaster and WDM incomplete1 Toaster Samples *(continued)*

WDM Function	Equivalent KMDF Function	Comments
ToasterDispatchPnP-Complete	None	KMDF driver uses defaults.
ToasterDispatch-Power	None	KMDF driver uses defaults.
Toaster System Control	None	WDM function passes WMI IRPs to the next lower driver. KMDF driver uses defaults.
None	ToasterEvtIoRead	WDM function handles and completes read request. KMDF function is a stub.
None	ToasterEvtIoWrite	WDM function handles and completes write requests. KMDF function is a stub.
None	ToasterEvtIoDeviceControl	WDM function handles and completes device I/O control request. KMDF function is a stub.

9.2 Macros Used in KMDF Samples

Many of the KMDF samples use two macros that might be unfamiliar:

- **PAGED_CODE**
- **UNREFERENCED_PARAMETER**

The **PAGED_CODE** macro causes the driver to assert if the function is called at **IRQL DISPATCH_LEVEL** or higher. The macro is defined in **ntddk.h** and takes no arguments. It works only in a checked build.

The **UNREFERENCED_PARAMETER** is defined in the standard WDK header file **ntdef.h**, which is included by **ntddk.h**. It disables compiler warnings about unreferenced parameters. It can be used in any Kernel Mode Driver.

9.3 KMDF Driver Structure and Concepts

KMDF drivers are object-oriented, event-driven drivers that link dynamically with the Kernel Mode Driver Framework at run time. This discussion provides a brief overview of KMDF concepts.

The KMDF object model defines object types to represent common driver constructs. Each object exports methods (functions) and properties (data) that drivers can access and is associated with object-specific events, which drivers can support by providing event callbacks. The objects themselves are opaque to the driver. KMDF and the driver instantiate the objects that are required to service the device. The driver provides callbacks for the events for which the KMDF defaults do not suit its device and calls methods on the object to get and set properties and perform any additional actions. Thus, a KMDF driver consists of a **DriverEntry** function, callback routines for events that affect the driver or its devices, and whatever utility functions the driver requires.

All KMDF drivers create a **WDFDRIVER** object to represent the driver and a **WDFDEVICE** object to represent each device that the driver supports. Most drivers also create one or more **WDFQUEUE** objects to represent the driver's I/O queues. KMDF places I/O requests into the queues until the driver is ready to handle them.

Drivers can create additional objects as their device hardware and driver features require. KMDF objects are organized hierarchically, with the **WDFDRIVER** object as the root. The object hierarchy defines the object's lifetime—each object is deleted when its parent is deleted.

All KMDF objects are created in the same way, by using KMDF-defined initialization functions and an object creation method. Any KMDF object has one or more driver-defined object context areas, in which the driver can store data that is specific to that particular instance of the object.

The following discussions provide more information on object creation and context areas and on I/O queues and requests, which are fundamental to KMDF drivers.

9.3.1 Object Creation

To create a KMDF object, a driver follows these steps:

1. Initialize the configuration structure for the object, if one exists.
2. Initialize the attributes structure for the object, if necessary.
3. Call the creation method to create the object.

The object configuration structure and the object attributes structure supply basic information about the object and how the driver uses it. All object types have the same attributes structure, but the configuration structure for each type of object is different and some objects do not have one.

The configuration structure holds pointers to object-specific information, such as the driver's event callback functions for the object. The driver fills in this structure and then passes it to the framework when it calls the object creation method. The framework uses the information from the configuration structure to initialize the object. For example, the **WDFDRIVER** object contains a pointer to the driver's **EvtDriver-DeviceAdd** callback function, which KMDF invokes when a Plug and Play add-device event occurs.

KMDF defines functions named **WDF_Object_Config_INIT** to initialize the configuration structures, where **Object** represents the name of the object type. Not all object types have configuration structures or the corresponding initialization functions.

The object attributes structure (**WDF_OBJECT_ATTRIBUTES**) specifies attributes that are common to all objects:

- Callbacks to handle object cleanup and destruction.
- The interrupt request level (**IRQL**) at which the objects' callback functions are invoked and its locks are held.
- An object context area.
- Information about the context area, such as its size and type.

KMDF defines the following for use in initializing object attributes:

- **WDF_OBJECT_ATTRIBUTES_INIT**
- **WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE**
- **WDF_OBJECT_ATTIBUTES_SET_CONTEXT_TYPE**

The **WDF_OBJECT_ATTRIBUTES_INIT** function sets values for synchronization and execution level, which determine which of the driver's callbacks KMDF invokes concurrently and the highest IRQL at which they can be called. The two context-type initialization macros set information about the object's context area, which is described in the next section.

Although attributes can apply to any type of object, the defaults are typically acceptable for objects for which the driver does not define a context area. To accept the defaults, a driver specifies **WDF_NO_OBJECT_ATTRIBUTES**, in which WDK defines a NULL.

After initializing the object's configuration structure and attributes, the driver creates an instance of the object by calling the creation method for the object type with pointer to the attributes structure and any other object-type-specific parameters. Creation methods are all named **WdfObjectCreate**, where **Object** indicates the type of object. The creation method returns a handle to the created object. The driver subsequently uses the handle to refer to the object.

9.3.2 Object Context Area

Every instance of an object can have one or more object context areas. The object context area is a driver-defined storage area for data that is related to that particular instance, such as a driver-allocated event. The driver determines the size and layout of the object context area. For a device object, the object context area is the equivalent of the WDM device extension.

A driver initializes the context area and specifies its size and type when it creates the object. When KMDF creates the object, it allocates memory for the context areas from the nonpaged pool and initializes them according to the driver's specification. When KMDF deletes the object, the context areas are deleted along with the object.

The context area is considered part of the object, which is opaque to drivers. Therefore, the driver must use an accessor method to get a pointer to the context area. Each context area has its own accessor method, and KMDF provides macros to create these methods.

For the driver, defining and initializing a context area is a multistep process. First, the driver defines a data structure that describes the context area. This definition typically appears in a header file.

Next, the driver declares the type of the context area, by using either the **WDF_DECLARE_CONTEXT_TYPE** or **WDF_DECLARE_CONTEXT_TYPE_WITH_NAME** macro. These macros associate a type with the context area and create a named accessor method that returns a pointer to the context area. **WDF_DECLARE_CONTEXT_TYPE_WITH_NAME** assigns a driver-specified name to the accessor method. **WDF_DECLARE_CONTEXT_TYPE** assigns the default name **WdfObjectGet_ContextStructure**, where **ContextStructure** is the name of the context structure. This step, too, can be performed in a header file.

Finally, the driver associates the context area with a specific instance of an object. To do so, the driver initializes the object attribute structure with information about the context area by using the **WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE** or **WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE** macro. The **WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE** macro records information about the context area in an attributes structure, which the driver later supplies when it creates the object. **WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE** combines the actions of **WDF_OBJECT_ATTRIBUTES_INIT** and **WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE**—that is, it initializes the attribute structure with settings for synchronization scope and execution level in addition to information about the context. The driver passes the resulting attribute structure when it calls the creation method for the object.

9.3.3 I/O Queues

KMDF I/O queues manage requests that are targeted at the driver. A driver typically creates one or more queues, each of which can accept one or more types of request. A driver can have any number of queues, and they can all be configured differently. Both KMDF and the driver itself can queue I/O request.

The most important characteristics of any queue are the types of requests it accepts, the way in which it dispatches those requests, and whether KMDF handles power management for it.

A queue can accept one or more types of requests. For example, a driver might have one queue for read and write request and another that accepts only device I/O control requests.

The dispatch method determines how many requests the driver services at a given time. Queues can dispatch requests sequentially, in parallel, or manually. A sequential queue dispatches one request at a time. A parallel queue dispatches requests as soon as they arrive. A manual queue does not dispatch requests automatically; the driver must call a method on the queue each time it is ready to handle another request.

By default, all KMDF I/O queues are power managed, which means that KMDF handles starting and stopping the queue according to the power management state of the device and the system. In addition, KMDF can use an empty queue as a cue to start its idle-device timer.

The driver uses a **WDF_IO_QUEUE_CONFIG** structure to configure a queue. For each queue, the driver can specify

- The types of I/O requests that are placed in the queue.
- The dispatch method for the queue.
- The power management options for the queue.
- The I/O event callback functions registered to handle I/O requests from the queue.
- Whether the queue accepts requests that have a zero-length buffer.

WDF supplies two functions to initialize the **WDF_QUEUE_CONFIG** structure:

- **WDF_IO_QUEUE_CONFIG_INIT**, which configures a power-managed queue with a driver-specified dispatch method.
- **WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE**, which configures a power-managed queue with a driver-specified dispatch method and designates it as the driver's default queue, into which KMDF places all read, write, and device I/O control requests for which no other queue is configured.

Like all other objects, queues have attributes. In many cases, the default attributes are acceptable for queues. A driver might override the defaults for one of the following reasons:

- To create one or more queue-specific context areas in which to save data specific to the queue.
- To specify a callback function to be invoked when the queue is deleted.
- To specify a parent object for the queue. By default, the parent object is the device object.

9.3.4 I/O Requests

The **WDFREQUEST** object represents the **IRP** sent by the application that initiated the I/O request. Like other KMDF objects, the **WDFREQUEST** object is opaque to driver writers and is managed by the framework. Most KMDF drivers never directly see the underlying **IRP**.

When KMDF calls one of the driver's I/O event callback functions, it passes a handle to a **WDFREQUEST** object, along with additional information that the driver might require to handle the request. The **WDFREQUEST** object encapsulates the information passed in the original **IRP**.

KMDF drivers that support buffered or direct I/O can use the buffer passed by the originator of the I/O request or can use a **WDFMEMORY** object that represents the output buffer. Using a **WDFMEMORY** object is simpler and requires less code because the framework handles all validation and addressing issues. For example, the handle to the **WDFMEMORY** object contains the size of the buffer, thus ensuring that buffer overruns do not occur.

9.4 A Minimal KMDF Driver: The Simple Toaster

The **Simple Toaster** sample provided in **toaster\func\simple** is a minimal, software-only function driver. It creates a driver object, a device object, a device interface, and a single I\O queue. The driver handles read, write, and device I/O control requests that are targeted at its device.

This minimal driver includes the following functions:

- A **DriverEntry** routine, which creates the driver object.
- An **EvtDriverDeviceAdd** event callback, which creates the device object, a device interface, and a default I/O queue.
- I/O callback functions for read, write, and device I/O control requests.

The driver does not manage any physical hardware, so no code to support Plug and Play or power management is required; the driver uses the WDF defaults.

9.4.1 Creating a WDF Driver Object: DriverEntry

Every KMDF driver must have a **DriverEntry** routine. The **DriverEntry** routine is the first driver function called when the driver is loaded. The KMDF **DriverEntry** routine has the same prototype as that of a WDM driver:

```
NTSTATUS
   DriverEntry(
      IN PDRIVER_OBJECT       DriverObject,
      In PUNICODE_STRING      RegistryPath
      );
```

The **DriverEntry** routine performs the following tasks:

- Creates a driver object (**WDFDRIVER**), which represents the loaded instance of the driver in memory. In effect, creating this object "registers" the driver with KMDF.
- Registers the driver's **EvtDriverDeviceAdd** callback. KMDF calls this function during device enumeration.
- Optionally initializes event tracing for the driver.
- Optionally allocates resources that are required on a driver-wide (rather than per-device) basis.

The following shows the **DriverEntry** function for the **Simple Toaster** sample:

```
NTSTATUS
DriverEntry(
   IN PDRIVER_OBJ      DriverObject,
   IN PUNICODE_STRING  RegistryPath
   )
{
   NTSTATUS          status = STATUS_SUCCESS;
   WDF_DRIVER_CONFIG   config;

   KdPrint(("Toaster Function Driver Sample-"
         Driver Framework Edition.\n));
   KdPrint(("Built %s %s\n", __DATE__, __TIME__));
```

```
    WDF_DRIVER_CONFIG_INIT(
        &config,
        ToasterEvtDeviceAdd
        );

    //
    // Create a framework driver object.
    //
    Status = WdfDriverCreate(
        DriverObject,
        RegistryPath,
        WDF_NO_OBJECT_ATTRIBUTES    // Driver Attributes
        &config,                    // Driver Config Info
        WDF_NO_HANDLE
        );

    if ( !NT_SUCCESS(status)) {
        KdPrint((“WdfDriverCreate failed with”
                “status 0x%x\n”, status));
    }

    return status;
}
```

Before creating the **WDFDRIVER** object, the driver must initialize a driver-object configuration structure **(WDF_DRIVER_CONFIG)** by using the **WDF_DRIVER_CONFIG_INIT** function. The function zeroes the structure and then initializes it with a pointer to the driver's **EvtDriverDeviceAdd** callback function, which is named **ToasterEvt-DeviceAdd**. KMDF calls this function during device enumeration, when it handles an add-device request that is targeted at the driver.

After setting up the configuration structure, **DriverEntry** calls **WdfDriverCreate** to create the **WDFDRIVER** object, passing as parameters the **DriverObject** and **RegistryPath** that were supplied to it, a pointer to the driver object attributes, and a pointer to the filled-in driver configuration structure. The framework's default attributes are acceptable for the **Simple Toaster's WDFDRIVER** object, so the sample specifies **WDF_NO_OBJECT_ATTRIBUTES**, which KMDF defines as **NULL**.

WdfDriverCreate can optionally return a handle to the created **WDFDRIVER** object. The **Simple Toaster** driver does not require a local copy of this handle, so instead of passing a location to receive the handle, it passes **WDF_NO_HANDLE**, which is a null pointer.

A WDM driver would typically save pointers to the **DriverObject** and **RegistryPath**, but a KMDF driver does not require them because KMDF maintains this information on behalf of the driver.

9.4.2 Creating the Device Object, Device Interface, and I/O Queue: EvtDriverDeviceAdd

Every KMDF driver that supports Plug and Play must have an **EvtDriverDeviceAdd** callback function, which is called each time the system enumerates a device that belongs to the driver. This callback performs actions required at device enumeration, such as the following:

- Creates and initializes a device object (**WDFDEVICE**) and corresponding context areas.
- Sets entry points for the driver's Plug and Play and power management callbacks.
- Creates a device interface.
- Configures and creates one or more I/O queues.
- Creates an interrupt object, if the device controls physical hardware that generates interrupts.

The **EvtDriverDeviceAdd** function is called with two parameters: a handle to the **WDFDRIVER** object that the driver created during **DriverEntry** and a handle to a **WDFDEVICE_INIT** object.

The **Simple Toaster** does not control hardware, so it does not set Plug and Play or power management callbacks, nor does it create an interrupt object. Its **EvtDriverDeviceAdd** callback creates the device object and context area, device interface, and a single default I/O queue. The following shows the source code for this function:

```
NTSTATUS
ToasterEvtDeviceAdd(
    IN WDFDRIVER         Driver,
    IN PWDFDEVICE_INIT   DeviceInit
    )
{
    NTSTATUS                  status = STATUS_SUCCESS;
    PFDO_DATA                 fdoData;
    WDF_IO_QUEUE_CONFIG       queueConfig;
    WDF_OBJECT_ATTRIBUTES     fdoAttributes;
    WDFDEVICE                 hDevice;
    WDFQUEUE                  queue;
    UNREFERENCED_PARAMETER(Driver);

    PAGED_CODE();
    KdPrint((“ToasterEvtDEviceAdd called\n”));
    //
    // Initialize attributes and a context area for the
    // device object.
    //
    //
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE
(&fdoAttributes, FDO_DATA);

    //
    // Create a framework device object.
    status = WdfDEviceCreate(&DeviceInit, &fdoAttributes,
        &hDevice);
    if(!NT_SUCCESS(status)) {
        KdPrint((“WdfDeviceCreate failed with status code”
                “0x%x\n”, status));
    return status;
}

// Get the device context by using the accessor function
// specified in the WDF_DECLARE_CONTEXT_TYPE_WITH_NAME
// macro for FDO_DATA.
//
fdoData = ToasterFdoGetData(hDevice);

//
// Create device interface.
//
status = WdfDEviceCreateDeviceInterface(
        hDevice,
        (LPUID) &GUID)DEVINTERFACE_TOASTER,
        NULL    // Reference String
        );

if(!NT_SUCCESS(status)) {
    KdPrint((“WdfDeviceCreateDeviceInterface failed
            “0x%x\n”, status));
    return status;
}

//
// Configure the default I/O queue.
//
```

```
WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&queueConfig,
   WdfIoQueueDispatchParallel);

// Register I/O callbacks for IRP_MJ_READ, IRP_MJ_WRITE,
// and IRP_MJ_DEVICE_CONTROL request.

queueConfig.EvtIoRead = ToasterEvtIoRead;
queueConfig.EvtIoWrite = ToasterEvtIoWrite;
queueConfig.EvtIoDeviceControl =
   ToasterEvtIoDeviceControl;

// Create the queue.

status = WdfIoQueueCreate (
   hDevice,
   &queueConfig,
   WDF_NO_OBJECT_ATTRIBUTES,
   &queue
   );

if(!NT_SUCCESS (status)) {
   KdPrint((“WdfIoQueueCreate failed 0x%x\n, status));
   return status;
}

return status;
}
```

The following discussions step through the actions of this function.

9.4.3 Device Object and Device Context Area

The first task of the **ToasterEvtDeviceAdd** callback is to initialize the context area and attributes for the **WDFDEVICE** object. The context area for this device object is a structure of type **FDO_DATA**, which is defined as follows in the header file **toaster.h**:

```
typedef struct _FDO_DATA
{
   WDFWMIINSTANCE             WmiDeviceArrivalEvent;
   BOOLEAN                    WmiPowerDeviceEnableRegistered;
   TOASTER_INTERFACE_STANDARD   BusInterface;
} FDO_DATA, *PFDO_DATA;

WDF_DECLARE_CONTEXT_TYPE_WITH(FDO_DATA, ToasterFdoGetData)
```

As the example shows, the header file defines the context area and then invokes the **WDF_DECLARE_CONTEXT_TYPE_WITH_NAME** macro. This macro creates an accessor method that is associated with a context type. Thus, when the **ToasterEvtDeviceAdd** function is called, the accessor method **ToasterFdoGetData** has already been created to read and write a context area of type **FDO_DATA**.

To associate the named context area with an object, the driver must initialize the object's attribute structure with information about the context area. The **ToasterEvtDeviceAdd** function invokes the **WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE** macro to do this:

```
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&fdoAttributes,
  FDO_DATA);
```

This macro performs the following tasks:

- Initializes **fdoAttributes**, which is a **WDF_OBJECT_ATTRIBUTES** structure.
- Sets pointers to the name and length of the context area and to the context area itself in the **WDF_OBJECT_ATTRIBUTES** structure.

The variable **fdoData**, of type **PFDO_DATA**, is defined to hold a pointer to the context area.

Next, the driver creates the **WDFDEVICE** object by calling **WdfDeviceCreate**, passing as parameters the addresses of the **WDFDEVICE_INIT** and **WDF_OBJECT_ATTRIBUTES** structures and a location to receive a handle to the create object:

```
status = WdfDeviceCreate(&DeviceInit, &fdoAttributes,
  &hDevice);
```

The framework allocates the **WDFDEVICE_INIT** structure, which is opaque to the driver writer. This object supports several methods that a driver can use to initialize device and driver characteristics, including the type of I/O that the driver supports, the device name, and a security descriptor definition language (SDDL) string for the device, among others. (These settings correspond to the device characteristics fields of the WDM device object, which is familiar to WDM driver writers.) By default, KMDF sets the I/O type to buffered I/O. This default, along with all the others, is appropriate for the **Simple Toaster**, so the driver does not call any methods on this object.

WdfDeviceCreate creates a **WDFDEVICE** object that is associated with an underlying WDM device object, connects the device object to the device stack, and sets the appropriate flags and attributes. It returns a handle to the **WDFDEVICE** object in **hDevice**.

After creating the device object, the driver gets a pointer to the context area by calling the accessor method **ToasterFdoGetData**:

```
fdoData = ToasterFdoGetData(hDevice);
```

The **ToasterEvtDeviceAdd** function does not use the returned pointer; this statement appears only for demonstration purposes.

9.4.4 Device Interface

Every device that a user-mode application or system component opens must have an interface. A device interface can be created in any of three ways:

- A user-mode installation application can create the interface using **SetupDi** function.
- An **INF** can create the interface by including a **DDInstall.Interfaces** section.
- The driver can create the interface by calling **WdfDeviceCreateInterface**.

The following shows how the **Simple Toaster** driver creates an interface:

```
status = WdfDeviceCreateDeviceInterface (
         hDevice,
         (LPUID) &GUID_DEVINTERFACE_TOASTER,
         NULL
         );
If (!NT_SUCCESS (status)) {
   KdPrinte((“WdfDeviceCreateDeviceInterface failed”
         “0x%x\n”, status));
   return status;
}
```

The driver passes a handle to the device object, a pointer to a Globally Unique Identifier (GUID), and a pointer to an optional string. The GUID identifies the interface and is defined in the **driver.h** header file. The string enables the driver to distinguish two or more devices of the same interface class (that is, two or more devices that have identical GUIDs). The **Simple Toaster** driver passes **NULL** for the string.

9.4.5 Default I/O Queue

The **SimpleToaster** driver uses a default I/O queue, for which KMDF handles power management. The default queue receives all I/O requests for which the driver does not specifically configure another queue. In this case, the default queue receives all read, write, and device I/O control requests that are targeted at the **Simple Toaster** driver. Default queues do not receive create requests.

The driver configures the queue by using the **WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE** function, which sets initial values in a configuration structure for the queue:

```
WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE (&ioEvents,
       WdfIoQueueDispatchParallel);
```

The function takes two parameters: a pointer to a **WDF_IO_QUEUE_CONFIG** structure **(ioEvents)** and an enumerator of the **WDF_IO_QUEUE_DISPATCH_TYPE**, which indicates how to dispatch requests from the queue. By specifying **WdfIoQueueDispatch-Parallel**, the driver indicates that KMDF should dispatch I/O requests from the queue as soon as they arrive and that the driver can handle multiple requests in parallel.

After configuring the queue, the driver registers its callbacks for I/O events. The **Simple Toaster** supports only read, write, and device I/O control requests, so it sets callbacks for only these three events in the **ioEvents** configuration structure. It then calls **WdfIoQueueCreate** to create the **WDFQUEUE** object, as the following code shows:

```
ioEvents.EvtIoRead = ToasterEvtIoRead;
ioEvents.EvtIoWrite = ToasterEvtIoWrite;
ioEvents.EvtIoDeviceControl = ToasterEvtIoDeviceControl;

status = WdfIoQueueCreate (
       hDevice,
       &ioEvents,
       WDF_NO_OBJECT_ATTRIBUTES,
       &queue       // pointer to default queue
       );
```

When the driver calls **WdfIoQueueCreate**, it passes a handle to the **WDFDEVICE** object (**hDevice**) and a pointer to the **ioEvents** configuration structure. The driver accepts the default attributes for the queue, so it passes **WDF_NO_OBJECT_ATTRIBUTES**. The method returns a handle to the queue at **&queue**. By default, KMDF places all read, write, and device I/O requests targeted at the device into this queue.

9.4.6 Handling I/O Request: EvtIoRead, EvtIoWrite, EvtIoDeviceControl

A driver can include one or more of the following I/O callback functions to handle the I/O requests that are dispatched from its queues:

- **EvtIoRead**
- **EvtIoWrite**
- **EvtIoDeviceControl**
- **EvtIoInternalDeviceControl**
- **EvtIoDefault**

For each queue, the driver registers one or more such callbacks. When an I/O request arrives, KMDF invokes the callback that is registered for that type of request, if one exists. For example, when a read request arrives, KMDF dispatches it to the **EvtIoRead** callback. Write requests are dispatched to the **EvtIoWrite** callback, and so forth. KMDF calls the **EvtIoDefault** callback when a request arrives for which the driver has not registered another callback. (In some cases, **EvtIoDefault** is also called to handle create request.)

KMDF queues and dispatches only the request for which the driver configures a queue. If KMDF receives any other I/O requests targeted at the **Simple Toaster** driver, it fails the requests with **STATUS_INVALID_DEVICE_REQUEST**.

As the preceding section mentions, the **Simple Toaster** driver handles only read, write, and device I/O control requests, so it includes only the **EvtIoRead**, **EvtIoWrite**, and **EvtIoDeviceControl** callback functions.

When KMDF invokes these callbacks, it passes a handle to the **WDFQUEUE** object, a handle to the **WDFREQUEST** object that represents the I/O request, and one or more additional parameters that provide details about the request. The additional parameters vary depending on the specific callback.

If the driver supports buffered or direct I/O, it can use either the buffer passed by the originator of the request or a **WDFMEMORY** object that represents that buffer. Using a **WDFMEMORY** object is simpler and requires less code because the framework handles all validation and addressing issues. For example, the handle to the **WDFMEMORY** object contains the size of the buffer, thus ensuring that buffer overruns does not occur. The **Simple Toaster** driver uses this technique.

To get a handle to the **WDFMEMORY** object, the driver calls **WdfRequestRetrieveOutputMemory** for a read request and **WdfRequestRetrieveInputMemory** for a write request. Each of these methods creates a **WDFMEMORY** object that represents the corresponding buffer and is associated with the **WDFREQUEST** object.

To handle a read request, the driver then gets the data from its device and uses **WdfMemoryCopyFromBuffer** to copy the data from its internal buffer into the **WDFMEMORY** object that is associated with the request.

To handle a write request, the driver has three options:

- Calling **WdfMemoryCopyToBuffer** to copy data from the **WDFREQUEST** object to the driver's internal buffer, from which the driver can write to the device.
- Getting the buffer pointer from the request by calling **WdfRequest-RetrieveInputBuffer**, which also returns the number of bytes to write.
- Getting the buffer pointer and the number of bytes to write by calling **WdfMemoryGetBuffer**.

For most write requests, drivers should use **WdfMemoryCopyToBuffer** to copy data supplied in the I/O request (and now stored in the associated **WDFMEMORY** object) to the driver's output buffer. This function copies the data and returns an error if a buffer overflow occurs. Use the **WdfRequestRetrieveInputBuffer** and **WdfMemoryGetBuffer** methods only if you require the buffer pointer so that you can cast it to a structure, which might be necessary when handling a device I/O control request.

Initially, the names **WdfMemoryCopyFromBuffer** and **WdfMemoryCopyToBuffer** might appear somewhat confusing. Remember that the word "Memory" in the name means that the function acts on a **WDFMEMORY** object and copies data to or from a program buffer into that object. Thus, **WdfMemoryCopyFromBuffer** copies data from the driver's internal buffer into a **WDFMEMORY** object, so it is used for read requests. **WdfMemoryCopyToBuffer** copies data to the driver's internal buffer, so it is used for write requests.

When the driver has satisfied the I/O request, it calls **WdfRequestCompleteWithInformation**, which completes the underlying I/O request with the specified status and passes the number of bytes read or written.

The **Simple Toaster** sample's **EvtIoRead**, **EvtIoWrite**, and **EvtIoDeviceControl** callbacks are essentially stubs. Although the **ToasterEvtIoRead** function calls **WdfRequestRetrieveOutput-Memory** to get the output buffer and the **ToasterEvtIoWrite** function calls **WdfRequestRetrieveInputMemory** to get the input buffer, neither function reads, writes, or returns any data. Therefore, none of these I/O event callbacks is reproduced here.

9.5 Sample Software-Only Driver

The **Featured Toaster** sample extends the **Simple Toaster** sample by adding support for the following features:

- File create and close requests.
- Additional device object attributes.
- Plug and Play and power management events.
- Windows management instrumentation (WMI).

This sample is supplied in **Toaster\Func\Featured** and supports the same features as the **Featured2 Toaster** sample for WDM, which is provided in the WDK.

9.5.1 File Create and Close Requests

File object events occur when applications and kernel-mode components that open a handle to the device issue create, close, and cleanup requests on the device. Drivers that handle such requests must configure the device object with the appropriate callbacks.

To handle create events, a driver can either receive the events from a queue or can supply an event callback that is invoked immediately. The driver's options are the following:

- To be called immediately, the driver supplies an **EvtDeviceFile-Create** callback and registers it from the **EvtDriverDeviceAdd** callback by calling **WdfDeviceInitSetFileObjectConfig**.
- To configure a queue to receive the requests, the driver calls **WdfDeviceConfigureRequestDispatching** and specifies **Wdf-RequestTypeCreate**. If the queue is not manual, the driver must register an **EvtIoDefault** callback, which is called when a create request arrives.

Queuing takes precedence over the **EvtDeviceFileCreate** callback—that is, if the driver both registers for **EvtDeviceFileCreate** events and configures a queue to receive such requests, KMDF queues the requests and does not invoke the callback. KMDF does not queue requests to a default queue; the driver must explicitly configure a queue to receive them.

In a function or bus driver, if a create request arrives for which the driver has neither registered an **EvtDeviceFileCreate** callback function nor configured a queue to receive create request, KMDF opens a file object to represent the device and completes the request with **STATUS_SUCCESS**. Therefore, any function or bus driver that does not accept create or open requests from user-mode applications—and thus does not register a device interface—must register an **EvtDeviceFile-Create** callback that explicitly fails such requests. Supplying a callback to fail create requests ensures that a rogue user-mode application cannot gain access to the device.

If a filter driver does not handle create requests, KMDF by default forwards all create, cleanup, and close requests to the default I/O target (the next lower driver).

Filter drivers that handle create requests should perform whatever filtering tasks are required and then forward such requests to the default I/O target. If the filter driver completes a create request for a file object, it should set **AutoForwardCleanupClose** to **WdfFalse** in the file object configuration so that KMDF completes cleanup and close requests for the file object instead of forwarding them.

To handle file close and cleanup requests on a device handle, the driver supplies the **EvtFileClose** and **EvtFileCleanup** callbacks. It registers both of these callbacks in its **EvtDriverDeviceAdd** function.

9.5.1.1 Register EvtDeviceFileCreate and EvtFileClose Callbacks

The following code is excerpted from the **Featured Toaster's ToasterEvtDeviceAdd** function. It shows how the driver registers the **EvtDeviceFileCreate** and **EvtFileClose** callbacks.

```
WDF_FILEOBJECT_CONFIG_INIT(
       &fileConfig,
       ToasterEvtDeviceFileCreate,
       ToasterEvtFileClose,
       WDF_NO_EVENT_CALLBACK      // not interested in Cleanup
       );

WdfDeviceInitSetFileObjectConfig(DeviceInit,
                                 &fileConfig,
                                 WDF_NO_OBJECT_ATTRIBUTES);
```

As the example shows, the driver initializes a **WDF_FILEOBJECT_CONFIG** structure by calling the configuration function and supplying pointers to its **EvtDeviceFileCreate** and **EvtFileClose** callbacks. The driver does not implement the **EvtFileCleanup** callback, so it supplies **WDF_NO_EVENT_CALLBACK** instead of a third function pointer.

It then calls **WdfDeviceInitSetFileObjectConfig** to record these settings in the **WDFDEVICE_INIT** structure **(DeviceInit)**, which the framework uses later when it creates the device object.

The **EvtDeviceFileCreate** and **EvtFileClose** functions are stubs in the **Featured Toaster** driver, so they are not shown here.

9.5.2 Additional Device Object Attributes

The **Featured Toaster** sample sets three more attributes for the device object than the **Simple Toaster** sample does:

- Synchronization scope
- Execution level
- An **EvtCleanupCallback** function

9.5.2.1 Synchronization Scope

The synchronization scope determines which of the callbacks for a device or queue object KMDF calls concurrently and which it calls sequentially. The following are the possible scopes:

- **Device scope (WdfSynchronizationScopeDevice)** means that KMDF does not make concurrent calls to certain I/O event callbacks for the queue or file objects that are children of the device object.
- **Queue scope (WdfSynchronizationScopeQueue)** means that KMDF does not call certain I/O event callbacks concurrently for the individual queue objects that are children of the object.
- **No scope (WdfSynchronizationScopeNone)** means that KMDF does not acquire any locks and can call any event callback concurrently with any other event callback. This is the default value for the **WDFDRIVER** object, which is the root object.
- **Default scope (WdfSynchronizationScopeInheritFromParent)** means that the object uses the same scope as its parent object. This is the default value for all objects other than the **WDFDRIVER** object.

By setting synchronization scope to device level **(WdfSynchronization-ScopeDevice)**, a driver tells KMDF to synchronize calls to certain I/O event callbacks (such as **EvtIoRead**, **EvtIoWrite**, and so forth) for queue and file objects that are children of the device object so that only one such callback executes at any given time. In effect, the callbacks execute synchronously. Because only one callback runs at a time, the driver is not required to hold locks to protect data that the callbacks share, such as the device context area.

Drivers can use these KMDF synchronization techniques (sometimes called frameworks locking) to synchronize access to their own hardware and their internal data. However, drivers should not use these techniques when calling externally, particularly to WDM drivers, because KMDF might hold a lock when the WDM driver does not expect it. This problem can occur because of IRQL restrictions or when the WDM driver eventually calls back into the KMDF driver, which results in a deadlock.

9.5.2.2 Execution Level

The execution level indicates the IRQL at which KMDF should call the event callback for the object. A driver can select **PASSIVE_LEVEL (WdfExecutionLevelPassive)**, **DISPATCH_LEVEL (WdfExecution-LevelDispatch)**, or the default level **(WdfExecutionLevelDefault)**. If the driver sets the default execution level for an object, KMDF does not guarantee callbacks at a particular **IRQL**, but can call them at any **IRQL <= Dispatch_Level**. For example, KMDF would call the routines at **DISPATCH_LEVEL** if an upper-level driver or KMDF itself already holds a spin lock.

Setting dispatch execution level for an object does not force its call-backs to occur at **DISPATCH_LEVEL**. Instead, it indicates that the callbacks are designed to be called at **DISPATCH_LEVEL** and so do not take any actions that might cause a page fault. If such a callback requires synchronization, KMDF uses a spin lock, which raises **IRQL** to **DISPATCH_LEVEL**.

Selecting passive execution level means that KMDF calls the callbacks at **PASSIVE_LEVEL**, even if it must queue a work item to do so. This level is typically useful only for very basic drivers, such as the **Toaster** sample. The **Featured Toaster** driver specifies **WdfExecutionLevelPassive**, thus ensuring that certain callbacks are called at **PASSIVE_LEVEL** so it can use pageable data in all such callbacks.

Most drivers must coexist in a stack with WDM drivers and therefore should accept the default.

9.5.2.3 EvtCleanupCallback Function

KMDF calls the object's **EvtCleanupCallback** callback when the object is being deleted, so that the driver can perform any cleanup related to the object. Device objects are typically deleted during device removal processing, so the **EvtCleanupCallback** for a device object is not called unless the device is removed.

If the device explicitly takes out a reference on an object (by calling **WdfObjectReference**), it must implement **EvtCleanupCallback** for that object so that it can release the reference. This callback also typically deallocates memory buffers that the driver previously allocated on a per-object basis and stored in the object context area.

In many drivers, this callback is not required. Remember that KMDF destroys the child objects of each parent object before deleting the parent object itself. If you allocate a **WDFMEMORY** buffer for object-specific storage and ensure that the **ParentObject** field in the attributes structure of the buffer points to the device object, you can often avoid the necessity of deallocating the buffer because KMDF deletes it automatically

In the **Featured Toaster** sample, this function is simply a stub.

9.5.3 Setting Additional Device Object Attributes

The following listing shows how the **Featured Toaster** driver sets these additional attributes and creates the **WDFDEVICE_OBJECT**:

```
//
// Set the Synchronization scope to device so that only one Evt
// callback for this device is executing at any time. This
// eliminates the need to hold any lock to synchronize access
// to the device context area.
//
// By specifying passive execution level, driver ensures
// that the framework will never call the I/O callbacks
// at DISPATCH_LEVEL.
//

fdoAttributes.SynchronizationScope  =
  WdfSynchronizationScopeDevice;
fdoAttributes.ExecutionLevel   =  WdfExecutionLevelPassive;

//
// Set a context cleanup routine to clean up any resources
// that are not defined in the parent to this device. This
// cleanup routine will be called during remove-device
// processing when the framework deletes the device object.
//

fdoAttributes.EvtCleanupCallback  =
  ToasterEvtDeviceContextCleanup;

//
// DeviceInit is completely initialized. Now call the
// framework to create the device object.
//
```

```
status   = WdfDeviceCreate(&DeviceInit, &fdoAttributes,
  &device);
if(!NT_SUCCESS(status)) {
   KdPrint((“WdfDEviceCreate failed with Stats code 0x%x\n”,
         status));
   return status;
}
```

The driver sets the attributes in **fdoAttributes**, and then creates the **WDFDEVICE** object.

This page intentionally left blank

CHAPTER 10

PROGRAMMING PLUG AND PLAY AND POWER MANAGEMENT

A few drivers, like the **Simple Toaster** sample, can use the **WDF** default for Plug and Play and power management. Most drivers, however, must perform device-specific activities in response to Plug and Play and power management requests.

The **Featured Toaster** driver is the power policy manager for its device and supports wake signals and idling in a low-power state. To implement such support, the driver must

- Register callbacks for Plug and Play, power, and power policy events.
- Set power policy options for its device.
- Implement callbacks to handle wake-up when a wake signal arrives.
- Implement callbacks for power-up and power-down to support idling in a low-power state.

This chapter provides an introduction to these features as they are implemented in the **Featured Toaster** sample. The chapter briefly outlines the purpose of these functions. However, because the **Featured Toaster** sample does not drive physical hardware, all of the callbacks are essentially stubs.

10.1 Registering Callbacks

Plug and Play and power events are defined for the device object, so the driver must register them in its **EvtDriverDeviceAdd** callback when it creates the device object. Most of the Plug and Play and power callbacks are defined in pairs: One event occurs when the device is added or powered up, and the other occurs when the device is removed or powered down. The following are the two most commonly used pairs:

- **EvtDeviceD0Entry** and **EvtDeviceD0Exit**, which are called immediately after the device enters the working state and before it exits the working state.
- **EvtDevicePrepareHardware and EvtDeviceReleaseHardware**, which are called before the device enters the working state and after it exits the working state. Typically, drivers map and unmap hardware resources in these callbacks.

KMDF defines the **WDF_PNPPOWER_EVENT_CALLBACKS** structure to hold the Plug and Play and power settings and provides a function to record the information from that structure into the **WDFDEVICE_INIT** structure. It also includes a method to set the Plug and Play and power management callbacks. Thus, to register its Plug and Play and power management callbacks, a driver

1. Calls **WDF_PNPPOWER_EVENT_CALLBACKS_INIT** to initialize a **WDF_PNPPOWER_EVENT_CALLBACKS** structure.
2. Sets entry points in the **WDF_PNPPOWER_EVENT_CALLBACKS** structure for its Plug and Play and power management event callbacks.
3. Calls **WdfDeviceInitSetPnpPowerEventCallbacks**, passing pointers to the **WDFDEVICE_INIT** structure and the completed **WDF_PNPPOWER_EVENT_CALLBACKS** structure.

A driver that manages power policy for its devices also must provide callback functions for power policy events, such as arming and disarming wake signals. The following are the power policy event callbacks:

- **EvtDeviceArmWakeFromS0**, which is called to arm the device to wake the system from **S0**, the system working state.
- **EvtDeviceDisarmWakeFromS0**, which is called to disarm wake signals from **S0**.
- **EvtDeviceWakeFromS0Triggered**, which is called when the system is in **S0** and the wake signal is triggered.
- **EvtDeviceArmWakeFromSx**, which is called to arm the device to wake the system from one of the lower-powered system sleep states.

- **EvtDeviceDisarmWakeFromSx**, which is called to disarm the wake signal, set by the **EvtDeviceArmWakeFromSx** callback.
- **EvtDeviceWakeFromSxTriggered**, which is called when the wake signal is triggered and the system is not in the working state.

Like the power management callbacks, the power policy event callbacks are required to initialize the device object. KMDF defines a structure and function that the driver can use to record this information in the **DEVICE_INIT** structure. The driver proceeds as follows:

1. Calls **WDF_POWER_POLICY_EVENT_CALLBACKS_INIT** to initialize a **WDF_POWER_POLICY_EVENT_CALLBACKS** structure.
2. Sets entry points for the power policy event callbacks that are supported by the driver.
3. Calls **WdfDeviceInitSetPowerPolicyEventCallbacks** with pointers to the **WDFDEVICE_INIT** structure and the completed **WDF_POWER_POLICY_EVENT_CALLBACKS** structure.

After creating the device object, the driver sets addition power policy options.

10.1.1 Sample Code to Register Plug and Play and Power Callbacks

The following code is excerpted from the **Featured Toaster's EvtDriverDeviceAdd** callback (**ToasterEvtDeviceAdd**). It shows how to register the Plug and Play, power management, and power policy callbacks.

```
//
// Initialize the pnpPowerCallbacks structure. Callback events
// for PNP and Power are specified here. If you don't supply any
// callbacks, KMDF will take appropriate default actions based on
// whether DeviceInit is initialized as an FDO, a PDO, or a
// filter device object.
//

WDF_PNPPOWER_EVENT_CALLBACKS_INIT (&pnpPowerCallbacks);

//
// Register PNP callbacks.
//
```

```
pnpPowerCallbacks.EvtDevicePrepareHardware =
       ToasterEvtDevicePrepareHardware;
pnpPowerCallbacks.EvtDeviceReleaseHardware =
       ToasterEvtDeviceReleaseHardware;

//
// Register Power callbacks.
//

pnpPowerCallbacks.EvtDeviceD0Entry = ToasterEvtDeviceD0Entry;
pnpPowerCallbacks.EvtDeviceD0Exit = ToasterEvtDeviceD0Exit;

WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit,
       &pnpPowerCallbacks);

//
// Register power policy event callbacks so that driver is called
// to arm/disarm the hardware to handle wait wake and
// when the wake event is triggered by the hardware.
//

WDF_POWER_POLICY_EVENT_CALLBACKS_INIT(
       &powerPolicyCallbacks);

//
// This group of three callbacks allows this sample driver to
// arm the device for wake from the S0 or Sx state. We don't
// differentiate between S0 and Sx state.
//

powerPolicyCallbacks.EvtDeviceArmWakeFromS0 =
       ToasterEvtDeviceWakeArm;
powerPolicyCallbacks.EvtDeviceDisarmWakeFromS0 =
       ToasterEvtDeviceWakeDisarm;
powerPolicyCallBack.EvtDeviceWakeFromS0Triggered =
       ToasterEvtDeviceWakeTriggered;
powerPolicyCallbacks.EvtDeviceArmWakeFromSx =
       ToasterEvtDeviceWakeArm;
powerPolicyCallbacks.EvtDeviceDisarmWakeFromSx =
       ToasterEvtDeviceWakeDisarm;
powerPolicyCallbacks.EvtDeviceWakeFromSxTriggered =
       ToasterEvtDeviceWakeTriggered;

//
// Register the power policy callbacks.
//
```

```
WdfDeviceInitSetPowerPolicyEventCallbacks(DeviceInit,
       &powerPolicyCallbacks);

//
// Additional code to initialize other features such as
// device file callbacks, the device context area, and
// other device object attributes is shown in the
// previous discussions
//
//
// Call the framework to create the
// device and attach it to the lower stack.
//

status = WdfDeviceCreate(&DeviceInit, &fdoAttributes, &device);
if (!NT_SUCCESS(status)) {
   KdPrint((“WdfDeviceCreate failed with Status Code 0x%x\n”,
      status));
   return status;
   }
```

The **Featured Toaster** driver sets callbacks for the **EvtDevicePrepareHardware**, **EvtDeviceReleaseHardware**, **EvtDeviceDOEntry**, and **EvtDeviceDOExit** events in **pnpPowerCallbacks**, which is a **WDF_PNPPOWER_EVENT_CALLBACKS** structure. It then calls **WdfDeviceInitSetPnpPowerEventCallbacks** to record this information in **DeviceInit**, which is a **WDFDEVICE_INIT** structure.

The **Featured Toaster** driver also sets callbacks for **EvtDevicePrepareHardware** and **EvtDeviceReleaseHardware**, in which a driver can map and unmap resources. However, because this driver does not operate physical hardware, these functions are essentially stubs.

The driver sets callbacks for power policy in **powerPolicyCallbacks**, which is a **WDF_POWER_POLICY_EVENT_CALLBACKS** structure. The **Featured Toaster** device requires the same driver actions regardless of the system state in which wake-up is armed, disarmed, or triggered. Therefore, the **Featured Toaster** sample provides only three callbacks for the six possible power policy events:

- **ToasterEvtDeviceWakeArm** arms the device for wake-up.
- **ToasterEvtDeviceWakeDisarm** disarms the device for wake-up.
- **ToasterEvtDeviceWakeTrigger** handles the wake-up signal.

The driver calls **WdfDeviceInitSetPowerPolicyEventCallbacks** to record these callbacks in the **DeviceInit** structure. Finally, it passes the **DeviceInit** structure in its call to **WdfDeviceCreate** to create the **WDFDEVICE** object. (The excerpt shown here omits the code that registers file object events and sets additional device object attributes; this code was shown previously.)

10.2 Managing Power Policy

After the driver creates the device object, it sets additional power policy options related to idling and wake signals before its **EvtDriver-DeviceAdd** callback returns.

The idle settings include the following:

- Idle time-outs, so that KMDF can transition the device out of the working state when it is not being used.
- The power state in which to idle the device.
- A Boolean that indicates whether a user can change the idle settings.

The **WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT** function initializes the structure that holds the idle settings. The driver passes a pointer to the structure and an enumerator that describes whether the device can generate a wake signal when the system is in **SO** (the fully powered, operational state). The **IdleCannotWakeFromSO** enumerator indicates that the device and driver do not support wake-up for **SO**. **IdleCanWakeFromSO** indicates that the device and driver support wake-up from **SO**. USB drivers that support Selective Suspend specify **IdleUsbSelectiveSuspend**.

The driver also sets a 60-second idle time-out, which indicates that the framework will attempt to put the device in a low-power state when it has been idle for more than 60 seconds.

By default, KMDF puts the driver in state **D3** when idle and enables the user to control the idle settings. A driver can change these defaults by setting values for the **DxState** field and the **UserControlOfIdleSettings** field in the **WDF_DEVICE_POWER_POWER_IDLE_SETTINGS** structure.

To register these settings, the driver then calls the **WdfDEvice-AssignSOIdleSettings** method on the **WDFDEVICE** object.

Next, the driver sets the wake-up policy. To initialize the policy settings to default values, the driver uses the **WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS_INIT** function. The defaults provide for the following:

- The device can wake up the system, but a user can override this default through a property page in Control Panel. This setting is appropriate for drivers that use the framework's WMI support to provide a property page through which a user can select the devices that are enabled to wake up the system.
- The device is put in its lowest powered state (**D3**) when the system goes into a sleep state.

A call to **WdfDeviceAssignSxWakeSettings** registers these settings.

10.2.1 Code to Set Power Policy

The following code shows how the **Featured Toaster** sets these power policy settings. This code is excerpted from the **ToasterEvtDeviceAdd** function in the file **toaster.c**.

```
//
// Set the idle power policy to put the device in Dx if the
// device
// is not used for the specified IdleTimeout time. Since
// this is a
// virtual device, we tell the framework that we cannot wake it
// if we sleep in SO. The only way to bring the device to DO is
// if the device receives an I/O request from the system.
//

WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT(&idleSettings,
   IdleCannotWakeFromSO);
idleSettings.IdleTimeout = 60000;   // 60 secs idle timeout
status = WdfDeviceAssignSOIdleSettings(device, &idleSettings);
if(!NT_SUCCESS(status)) {
   KdPrint(("WdfDEviceAssignSOIdleSettings failed 0x%x\n",
         status));
   return status;
   }
```

```
//
// Set the wait-wake policy.
//

WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS_INIT(
    &wakeSettings);
status = WdfDeviceAssignSxWakeSettings(device, &wakeSettings);
if(!NT_SUCCESS(status)) {
    //
    // We are probably enumerated on a bus that doesn't support
    // Sx-wake. Do not fail the device add just because we can't
    // support wait-wake.
    //

    KdPrint(("WdfDEviceAssignSxWakeSettings failed 0x%x\n",
            status));
    status = STATUS_SUCCESS;
}
```

The sample driver sets power policy idle characteristics and an idle time-out in the **idleSetting** structure, and then calls **WdfDEvice-AssignSOIdleSettings** method, passing as parameters this structure and the handle to the device object that is received when it created the device object. It then proceeds in a similar fashion to set the wake-up characteristics.

If the call to **WdfDeviceAssignScWakeSettings** fails, the sample driver resets the failure status to **STATUS_SUCCESS**. The reason is that this method fails if the underlying bus does not support wake from **Sx** states. If the device is on such a bus, the driver should not fail the add-device request. Instead, it simply resets the failure status and continues processing.

10.3 Callbacks for Power-Up and Power-Down

Drivers implement the **EvtDeviceDOEntry** and **EvtDeviceDOExit** callbacks to be notified when the device enters and exits the working state, respectively.

In a function driver (FDO), KMDF calls the **EvtDeviceDOEntry** callback after the device has entered the **DO** power state. The function driver's **EvtDeviceDOEntry** callback should initialize the device and perform any other tasks that are required each time the device enters the working state.

In a bus driver (PDO), KMDF calls the **EvtDeviceDOEntry** callback to put the device into the **DO** power state. (If you are familiar with WDM drivers, you probably remember that a request to power up a device must be handled by the bus driver first and then other drivers higher in the device stack. For a request to lower the power state, the opposite is true.) A bus driver's **EvtDeviceDOEntry** callback should put the device hardware in the **DO** state and perform any additional initialization that the device requires at this time.

The **EvtDeviceDOEntry** callback is called with a pointer to the **WDFDEVICE** object and a pointer to an enumerator that identifies the most recent power state of the device, before the transition to **DO** began.

KMDF calls the **EvtDeviceDOExit** callback any time the device must leave the **DO** state. The callback is passed a pointer to the **WDFDEVICE** object and a pointer to an enumerator that identifies the new power state. In a function driver (FDO), the callback should perform any final tasks that are required before the device changes power state. In a bus driver, the callback should change the power state of the physical hardware.

In the **Featured Toaster** sample, both of these functions are stubs.

10.4 Callback for Wake Signal Support

Drivers for devices that can generate wake signals implement the following callbacks to support wake-up:

- **EvtDeviceArmWakeFromSO** and **EvtDeviceDisarmWakeFromSO**
- **EvtDeviceWakeFromSOTriggered**
- **EvtDeviceArmWakeFromSx** and **EvtDeviceDisarmWakeFromSx**
- **EvtDeviceWakeFromSxTriggered**

To enable the wake signal, KMDF calls **EvtDeviceArmWakeFromSO** and **EvtDeviceArmWakeFromSx** while the device is in **DO**, but while power-managed I/O is stopped. Therefore, the driver does not receive I/O requests from power-managed queues while arming its device and so is not required to synchronize its I/O operations with these callbacks.

In the wake-arming callbacks, a function driver should handle any device-specific tasks that are required to enable the device to generate a wake signal. KMDF transitions the device out of the **DO** state soon after the wake signal is successfully enabled. The driver should not yet perform any tasks related to power state change; KMDF calls the driver's **EvtDeviceDOExit** function to do that.

The **EvtDeviceDisarmWakeFromSO** and **EvtDeviceDisarm-WakeFromSx** callbacks should undo any device-specific tasks that the driver performed to enable the wake signal for its device. KMDF calls the disarm callbacks after **EvtDeviceDOEntry** has returned the device to the **DO** state.

The **Featured Toaster** sample registers the **ToasterEvtDevice-WakeArm** function for both the **EvtDeviceArmWakeFromSO** and **EvtDeviceArmWakeFromSx** events. Likewise, it registers **Toaster-EvtDeviceWakeDisarm** for both the **EvtDeviceDisarmWakeFromSO** and **EvtDeviceDisarmWakeFromSx** events.

In the **Featured Toasted** sample, these callbacks are stubs.

If the device triggers a wake signal when the system is in **SO** or **Sx**, the **EvtDeviceWakeFromSOTriggered** and **EvtDeviceWakeFromSx-Triggered** events occur, respectively. KMDF calls these functions after the driver's **EvtDeviceDOEntry** callback and before the **EvtDevice-DisarmWakeFromSO** or **EvtDeviceDisarmWakeFromSx** callback. The wake-triggered callback should perform any tasks that are required after the wake signal is triggered, but should not disarm the wake signal.

The **Featured Toaster** sample registers the **ToasterEvtDevice-WakeTriggered** callback for both the **EvtDeviceWakeFrom-SOTriggered** and **EvtDeviceWakeFromSxTriggered** events. In the sample, this callback is a stub.

CHAPTER 11

PROGRAMMING WMI SUPPORT

This chapter covers the importance of using the Windows Management Instrumentation (WMI) in the development and utilization of a device driver.

By making your driver a Windows Management Instrumentation (WMI) provider, you can

- Make custom data available to WMI consumers.
- Permit WMI consumers to configure a device through a standard interface rather than a custom control panel application.
- Notify WMI consumers of driver-defined events without requiring the consumer to poll or send IRPs.
- Reduce driver overhead by collection and sending only requested data to a single destination.
- Annotate data and event blocks with descriptive driver-defined class names and optional descriptions that WMI clients can enumerate and display to users.

11.1 WMI Architecture

To support WMI, your driver registers as a WMI provider. A WMI provider is a Win32 dynamic-link library (DLL) that handles WMI requests and supplies WMI instrumentation data.

After your driver is registered as a WMI provider, WMI consumers then request data or invoke methods exposed by providers.

Query requests travel from user mode consumers down to WMI kernel mode service, which in turn send IRP requests to your driver.

For instance, when a WMI client requests a given data block, the WMI kernel component sends a query request to the driver to retrieve or set data. Figure 11.1 shows this data flow.

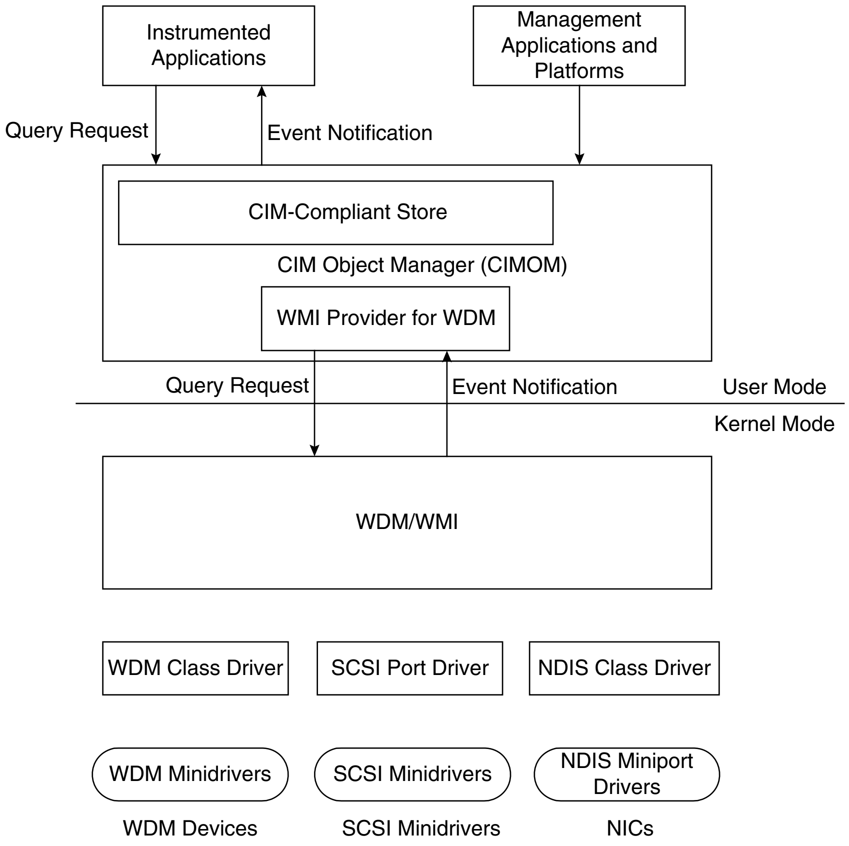

Figure 11.1 WMI Architecture Data Flow

11.2 Registering as a WMI Data Provider

A driver that supports WMI must register as a WMI data provider to make its data and event blocks available to WMI clients. A driver typically registers with WMI when starting its device, after the device has been initialized to the point that the driver can handle WMI IRPs. During the registration process, the driver passes WMI a pointer to its device object and information about the data and event blocks it supports.

A driver registers with WMI in two phases:

1. The driver calls **IoWMIRegistrationControl** with the action **WMIREG_ACTION_REGISTER** and a pointer to the device object passed to the driver's **AddDevice** routine.
2. The driver handles the **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request that WMI sends in response to the driver's **IoWMIRegistrationControl** call. The **Parameters.WMI.DataPath** member of the IRP is set to **WMIREGISTER** and **Parameters.WMI.ProviderId** is set to the driver's device object pointer. The driver supplies WMI with registration information about data and event blocks, either by using the WMI Library, or by handling the **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** requests.

11.3 Handling WMI Requests

All drivers must set a dispatch table entry point for a **DispatchSystemControl** routine to handle WMI requests. If a driver registers as a WMI data provider, it must handle all WMI requests. Otherwise, the driver must forward all WMI requests to the next lower driver.

All WMI IRPs have the major code **IRP_MJ_SYSTEM_CONTROL** and one of the following minor codes:

- **IRP_MN_REGINFO or IRP_MN_REGINFO_EX**—Queries or updates a driver's registration information after the driver has called **IoWMIRegistrationControl**.
- **IRP_MN_QUERY_All_DATA or IRP_MN_QUERY_SINGLE_INSTANCE**—Queries for all instances or a single instance of a given data block.
- **IRP_MN_CHANGE_SINGLE_ITEM, IRP_MN_CHANGE_SINGLE_INSTANCE**—Requests the driver to change a single item or multiple items in an instance of a data block.
- **IRP_MN_ENABLE_COLLECTION, IRP_MN_DISABLE_COLLECTION**—Requests the driver to start accumulating data

for a block that the driver registered as expensive to collect, or to stop accumulating data for such a block.

- **IRP_MN_ENABLE_EVENTS, IRP_MN_DISABLE_EVENTS**—Requests the driver to start sending notification of a given event if the event occurs while it is enabled, or to stop sending notification of such an event.
- **IRP_MN_EXECUTE_METHOD**—Requests the driver to execute a method associated with a data block.

The WMI kernel mode component sends WMI IRPs any time following a driver's successful registration as a WMI data provider, typically when a user mode data consumer has requested WMI information for a driver's device. If a driver registers as a WMI data provider by calling **IoWMIRegistrationControl**, it must handle each subsequent WMI request in one of the following ways:

- Call the kernel mode WMI library routine **WmiSystemControl** to handle requests concerning only blocks that do not use dynamic instance names, and that base static instance names on a single base name string or the device instance ID of a PDO.
- In its **DispatchSystemControl** routine, process and complete any such request tagged with the pointer to its device object that the driver passed in its call to **IoWMIRegistrationControl**, and forward other **IRP_MJ_SYSTEM_CONTROL** requests to the next lower driver.

11.4 WMI Requirements for WDM Drivers

A driver that handles IRPs registers with WMI as a data provider. System-supplied storage port drivers, class drivers, and NDIS protocol drivers fall into this category.

A driver that does not handle IRPs should simply forward WMI requests to the next-lower driver in the driver stack. The next-lower driver then registers with WMI and handles WMI requests on the first driver's behalf. For instance, SCSI miniport drivers and NDIS miniport drivers can register as WMI providers and supply WMI data to their corresponding class drivers.

A driver that supplies WMI data to a class or port driver must support the driver-type-specific WMI interfaces that are defined by the class or port driver. For example, a SCSI miniport driver must set **WmiData Provider** to TRUE in the **PORT_CONFIGURATION_INFORMATION** structure and handle **SRB_FUNCTION_WMI** requests from the SCSI port driver.

Similarly, a connection-oriented NDIS miniport driver that defines custom data blocks must support **OID_GEN_CO_SUPPORTED_GUIDS**; otherwise, NDIS maps those OIDs and status indications returned from **OID_GEN_SUPPORTED_LIST** that are common and known to NDIS to GUIDS defined by NDIS.

11.5 WMI Class Names and Base Classes

WMI class names are case-insensitive, must start with a letter, and cannot begin or end with an underscore. All remaining characters must be letters, digits, or underscores.

WMI client applications can access a driver's WMI class names and display them to users. Descriptive class names can help make classes more intuitive to use.

WMI class names must be unique within the WMI name space. Consequently, a driver's WMI class names cannot duplicate those defined by another driver.

To help prevent name collisions, a driver writer can define a driver-specific base class and derive all of the driver's WMI classes from that base class. The class name and base class name together are more likely to yield a unique name. For example, the following shows an abstract base class for a serial driver's data blocks:

```
// Serial drivers' base class for data blocks
[abstract]
class MSSerial {
}

// Example class definition for a data block
[
    // Class qualifiers
]
class MSSerial_StandardSerialInformation : MSSerial
{
    // Data items
}
```

Device-specific custom data blocks should include the manufacturer, model, and type of driver or device in the base class name. For example:

```
[abstract]
class Adaptec1542 {
}
class Adaptec1542_Bandwidth : Adaptec 1542 {
   // Data items
}
class Adaptec1542_Speed : Adaptec1542 {
   // Data items
}
```

WMI allows only one abstract base class in a given class hierarchy. Classes that define event blocks must derive from **WmiEvent**, which is an abstract base class, so the **abstract** qualifier cannot be used in a driver-defined base class for event blocks. Instead, derive a nonabstract base class from **WmiEvent**, then derive individual event classes from that base class. The following examples show class definitions from the schema of a serial port driver. Not that the **guid** values shown in these examples are placeholders. Each class definition must have a unique GUID generated by **guidgen.exe** or **uuidgen.exe** (which are included in the Microsoft Windows SDK).

```
// Standard class for reporting serial port information
// Class qualifiers
[WMI, guid("xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"),
Dynamic, Provider("WMIProv"),
WmiExpense(1),
Locale("MS\\0x409"),
Description("Description of class"]

// Class name
class Vendor_SerialInfo {
// Required items
   [key, read]
      string InstanceName;
   [read]
      boolean Active;
// Bytes sent on port
// Property qualifiers
   [read,
      WmiDataId(1),
      WmiScale(0),
      WmiComplexity(1),
      WmiVolatility(1000)]
   Description("Description of property");
// Data item
   uint64 BytesSent;
// Bytes received on port
   [read,
      write,
      WmiDataId(2),
      WmiScale(0),
      WmiVolatility(1000)]
   uint64 BytesReceived
// Who owns the port
   [read,
      WmiDataId(4),
      WmiScale(0),
      WmiVolatility(6000)]
   string Owner;
// Status bit array
   [read, write,
      WmiDataId(3),
      WmiScale(0)]
   byte Status[16];
// The number of items in the XmitBufferSize array
   [read,
      WmiDataId(5),
      WmiScale(0),
      WmiComplexity(1),
      Wmivolatility(1000)]
   uint32 XmitDescriptionCount;
// Array of XmitDescription classes
   [read,
      WmiDataId(6),
      WmiSizeIs("XmitDescriptionCount"),
      WmiScale(0),
      Wmicomplexity(1),
      WmiVolatility(1000)]
   Vendor_XmitDescriptor XmitBufferSize[];
}
```

The following is the class definition for the embedded class shown in the preceding example. Note that this class does not contain **InstanceName** or **Active** items.

```
// Example of an embedded class
[WMI, guid("xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"),
class Vendor_XmitDescriptor {
   [read, WmiDataId(1)] int32 DestinationIndex;
   [read, WmiDataId(2)] int32 DestinationTarget;
}
```

The following is a class definition for an event block. The class is derived from **WmiEvent**.

```
// Example of an event
[WMI, guid("xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"),
Dynamic, Provider("WMIProve"),
Locale("MS\\0x409),
Description("Serial Send Event"),
WmiExpense(1)]

class Vendor_SerialSentEvent : WMIEvent
{
// Required items
   [key, read]
      string InstanceName;
   [read]
      boolean Active;
```

11.6 Firing WMI Events

In the **Featured Toaster** sample driver, the **ToasterFireArrivalEvent** function shows how to generate a WMI event that has a dynamic instance name. The WMI event contains the name of the device model, which WMI records to log the arrival of this device. The driver calls this function from its **EvtDevicePrepareHardware** callback to indicate that the device has been configured. Parts of this function are standard WMI tasks and have little to do with KMDF. This discussion focuses on the KMDF-specific code:

```
NTSTATUS
ToasterFireArrivalEvent(
   IN    WDFDEIVE        Device
   )
{
   WDFMEMORY                   memory;
   PWNODE_SINGLE_INSTANCE      wnode;
   ULONG                       wnodeSize;
   ULONG                            wnodeDataBlockSize;
   ULONG                            wnodeInstanceNameSize;
   ULONG                            size;
   ULONG                            length;
   UNICODE_STRING                   deviceName;
   UNICODE_STRING                   modelName;
   NTSTATUS                         status;

   //
   // *NOTE*
   // WdfWmiFireEvent only fires single instance events at
   // the present, so continue to use this method of firing
   // events
   // *NOTE*
   //

   RtlInitUnicodeString(&modelName, L"Sonali\0\0");

   //
   // Get the device name.
   //

   status = GetDeviceFriendlyName(&Device, &memory);
   if(!NT_SUCCESS(status)) {
      return status;
   }

   RtlInitUnicodeString(&deviceName, (PWSTR)
      WdfMemoryGetBuffer(memory, NULL));

   //
   //Determine the amount of wnode information we need.
   //

   wnodeSize = sizeof(WNODE_SINGLE_INSTANCE);
   wnodeInstanceNameSize = deviceName.Length + sizeof(USHORT);
   wnodeDataBlockSize = modelName.Length + sizeof(USHORT);

   size = wnodeSize + wnodeInstanceNameSize +
   wnodeDataBlockSize;


   //
   // Allocate memory for the WNODE from NonPagedPool
   //
```

```
    wnode = ExAllocatePoolWithTag(NoPagedPool, size,
            TOASTER_POOL_TAG);

If(NULL != wnode) {
        RtlZeroMemory(wnode, size);
        wnode->WnodeHeader.BufferSize = size
        wnode->WnodeHeader.ProviderId =
            IoWMIDeviceObjectToProviderId(
            WdfDeviceWdmGetDeviceObject(Device));
        wnode->WnodeHeader.Version = 1;
        KeQuerySystemTime(&wnode->WnodeHeader.TimeStamp);

        RtlCopyMemory(&wnode->WnodeHeader.Guid,
            &TOASTER_NOTIFY_DEVICE_ARRIVAL_EVENT,
            Sizeof(GUID));

        //
        // Set flags to indicate that you are creating dynamic
        // instance names. This driver supports dynamic
        // instances because it can fire the events at any time.
        // If it used static instance names instead, it could only
        // fire events after WMI queries for IRP_MN_REGINFO,
        // which happens after the device has been started.
        // Note also that if you are firing an event after the
        // device is started, you should
        // check whether the event is enabled, because that
        // indicates that someone is interested in receiving the
        // event. Why waste system resources by firing an event
        // when nobody is interested?
        //

        wnode->WnodeHeader.Flags = WNODE_FLAG_EVENT_ITEM |
                    WNODE_FLAG_SINGLE_INSTANCE;

        wnode->OffsetInstanceName = wnodeSize;
        wnode->DataBlockOffset= wnodeSize + wnodeInstanceNameSize;
        wnode->SizeDataBlock = wnodeDataBlockSize;

//
        // Write the instance name.
        //
        size -= wnodeSize;
        status = WDF_WMI_BUFFER_APPEND_STRING(
           WDF_PTR_ADD_OFFSET(wnode, wnod->OffsetInstance),
           size,
           &deviceName,
           &length
           );

        //
        // Size was precomputed, so this should never fail.
        //

        ASSERT(NT_SUCCESS(status));


        //
        // Write the data, which is the model name as a
           string.
        //

        size = -= wnodeInstanceNameSize;
        WDF_WMI_BUFFER_APPEND_STRING(
          WDF_PTR_ADD_OFFSET(wnode, wnode->DataBlockOffset),
             size,
             &modelName,
             &length
             );

           //
           // Size was precomputed, so this should never fail.
           //

           ASSERT(NT_SUCCESS(status));

           //
           // Indicate the event to WMI. WMI will take care of
           // freeing the WMI struct back to pool.
           //

status = IoWMIWriteEvent(wnode);

if(!NT_SUCCESS(status)) {
   KdPrint((“IoWMIWriteEvent failed %x\n”, status));
   ExFreePool(wnode);
}
        }
        else {
            status = STATUS_INSUFFICIENT_RESOURCES;
        }

        //
        // Free the memory allocated by GetDeviceFriendlyName
        // function.
        //

        WdfObjectDelete(memory);
        return status;
    }
```

ToasterfireArrivalEvent first defines a model name for its device and then retrieves the friendly name for the device by calling the internal routine **GetDeviceFriendlyName**. **GetDeviceFriendlyName** takes a handle to the device object and returns a handle to a **WDFMEMORY** object that contains a buffer that holds the name.

The driver passes the returned **WDFMEMORY** object handle to **WdfGetMemoryBuffer** to retrieve a pointer to the buffer itself and calls **RtlUnicodeString** to copy the contents of the buffer into the Unicode string variable **deviceName**, as follows:

```
RtlInitUnicodeString(&deviceName,
    (PWSTR)WdfMemoryGetBuffer(memory, NULL));
```

The driver now constructs the WMI event by using standard WMI structures and routines (defined in **wmistr.h**) along with standard Windows device driver interfaces (DDIs), which are defined in **wdm.h** and **ntddk.h**. The driver allocates memory from the nonpaged pool for the **WNODE_SINGLE_INSTANCE** structure (**wnode**) by calling **ExAllocatePoolWithTag**. If memory allocation succeeds, the driver zero-initializes the **wnode** structure and then fills it in the usual way for WMI. To supply the WMI provider **ID**, the driver calls **IoWmiDeviceObjectToProviderId**, passing as a parameter the WDM device object returned by **WdfDeviceWdmGetDeviceObject**.

After the **wnode** structure is filled, the driver calls the **WDF_WMI_BUFFER_APPEND_STRING** function once to append the instance name to the **WNODE_SINGLE_INSTANCE** structure and a second time to append the model name.

Finally, the driver fires the event by calling **IoWMIWriteEvent**. WMI records the data and frees the memory allocated for the **WNODE_SINGLE_INSTANCE** structure.

11.7 Troubleshooting Specific WMI Problems

The following paragraphs cover some of the common problems found in setting up and using WMI.

11.7.1 Driver's WMI Classes Do Not Appear in the \root\wmi Namespace

The following steps cover how we would approach the troubleshooting of this particular problem.

1. Use **wmimofck driver.bmf** to check if the binary MOF file format is correct. Additional error messages may be found in **mofcomp.log**.
2. Check the system event log to see if the driver is returning a malformed **WMIREGINFO** data structure in response to the registration request.
3. Check that the driver is returning the correct values for **RegistryPath** and **MofResourceName** within the **WMIREGINFO** structure.
4. If the driver provides its MOF data in a separate file, check that the **MofImagePath** registry value for the driver is set correctly.
5. Check the WMI WDM provider log for errors.
6. Use **Mofcomp** to recompile and reload your MOF text file. For example, the command **mofcomp –N:root/wmi driver.mof** tries to recompile and reload any MOF data in the **driver.mof** file. Check to see what error messages **Mofcomp** generates in **mofcomp.log**. Note that if your MOF file uses preprocessor directives such as **#define**, you need to use the already preprocessed MOF file, and not the original source file. If the operation succeeds, it actually registers the new WMI class data with the system. You need to delete these classes (by using **Wbemtest**, for example) to test if your driver's MOF data is being read correctly.

If this step succeeds, the most likely problem is that the members of **WMIREGINFO**, such as **MofResourceName**, are specified incorrectly. Alternatively, the problem could be that your MOF file specifies a class derived from a base class that doesn't exist.

7. If the driver is using dynamic MOF data, check that the driver is receiving WMI IRP requests for the **MSWmi_MofData_GUID** GUID and that is completing the IRP successfully and with no error logged.

11.7.2 Driver's WMI Properties or Methods Cannot Be Accessed

Use the following steps if you cannot access the driver's WMI properties or methods:

1. Use **wmimofck driver.bmf** to check if the binary MOF file format is correct.
2. Check the system event log for errors.
3. Check the WMI WDM Provider log for errors.
4. Make sure the driver receives a WMI IRP whenever you use Wbemtest to query the driver's classes. If not, check that the specified GUID in the MOF file matches the GUID the driver is expecting. Also check that the driver is receiving the WMI registration request, it is succeeding, and the driver is registering the right GUIDs.
5. If the driver receives the IRP, ensure that the IRP is completed successfully, and that the driver is returning the right type of **WNOFE_XXX** structure.
6. If Wbemtest returns an error, click the More Information button and check the Description property for a description of the error.
7. For methods, check that your driver supports handling the **IRP_MN_QUERY_ALL_DATA** and **IRP_MN_QUERY_SIGLE_INSTANCE** requests for the method's GUID. WMI always performs one of those two requests before executing a method.

11.7.3 Driver's WMI Events Are Not Being Received

Use the following steps if the driver's WMI events are not being received:

1. Check the system event log for errors. For example, if the driver specifies a static event name when calling **IoWMIWriteEvent** but the driver did not register any static event names, this would produce an entry in the system event log.
2. Check the WMI WDM provider log for errors.

   Note: If the driver is sending an event reference, the driver should receive an **IRP_MN_QUERY_SINGLE_INSTANCE** request immediately after sending the event reference. If the driver does not receive the IRP, the **WNODE_EVENT_REFERENCE** structure may have been malformed. If the driver receives the IRP, it should be completing it with status **STATUS_SUCCESS**.
3. If the driver uses **IoWMIWriteEvent** to send the event or event reference, make sure the event structure (either **WNODE_SINGLE_INSTANCE** or **WNODE_EVENT_REFERENCE**) is filled out correctly. In particular, if the event GUID is registered for static instance names, make sure that the correct instance index and provider ID are provided. If the event GUID is registered for dynamic instance names, make sure the instance name is included when the event is sent. If using the **WNODE_EVENT_REFERENCE** structure to specify the event, check that **Wnode.Guid** matches **TargetGuid**.
4. If the driver uses **WmiFireEvent** to send the event, make sure the correct value is passed for the **Guid** and **InstanceIndex** parameters.

11.7.4 Changes in Security Settings for WMI Requests Do Not Take Effect

If the changes we made in security settings for WMI Request do not take effect, you should do the following.

Unload and reload the WMI WDM Provider. For WMI data blocks registered with the **WMIREG_FLAG_EXPENSIVE** flag, the provider keeps a handle open to the data blocks as long as there are consumers for that block. The new security settings will not take effect until the provider closes the handle. Unloading and reloading the provider makes sure the handle has been closed.

11.8 Techniques for Testing WMI Driver Support

There are several tools you can use to test WMI support in your driver:

- **Wbemtest**—The operating system includes the **Wbemtest** tool, which provides a GUI you can use to query for WMI classes and class instances, change property values, execute methods, and receive event notifications. Connect to the **root\wmi** namespace to test your driver support.
- **Wmic**—Microsoft Windows 7 operating system includes the **Wmic**, which provides a command shell you can use to issue WMI-related commands to test your driver.
- **Wmimofck**—The **wmimofck** command can be used to check the syntax of your binary MOF files. You can also use the **wmimofck –t** command to generate a VBScript file. You can use this script to test your driver's handling of WMI class instance queries. The **wmimofck –w** command generates Web pages that can test querying and setting classes, executing methods, and receiving events. Note that the Web pages do not support executing methods that use complex parameters or return values (such as an array of embedded classes). In such cases, you can use **Wbemtest** instead.

You can also test your driver's WMI support by writing a custom WMI client application, using the WMI user mode API. For more information about this user mode API, which allows applications to provide or consume WMI information, refer to the Windows Management Instrumentation information in the Microsoft Windows SDK documents.

A WMI client application performs the following tasks to test a driver:

1. It connects to WMI. To connect to WMI, the application calls the Component Object Model (COM) function, **CoCreateInstance**, to retrieve a pointer to the **IWbemLocator** interface. The application then calls the **IWbemLocator::ConnectServer** method to connect to WMI. From this call, the application receives a pointer to the **IWbemServices** interface.
2. It accesses information in the driver. To access information and to register for events, the application uses the methods of the **IWbemServices** interface.

11.8.1 WMI IRPs and the System Event Log

WMI errors that occur strictly in kernel mode are logged to the system event log. You can use the Event Viewer to examine the system event log.

The two main sources of such errors are malformed replies to WMI requests and incorrect parameters to event notifications. For example, if the driver returns a malformed **WMIREGINFO** data structure in response to an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request, the system will log that to the system event log. The system would also log an invalid call to **IoWMIWriteEvent** and **WmiFireEvent** to issue a WMI event notification.

11.8.2 WMI WDM Provider Log

WMI errors that occur while being handled by the WMI WDM provider (**wmiprov.dll**) are logged to the log file for the WMI WDM Provider, **wmiprov.log**. This is a text file that can be found in **%windir%\system32\wbem\logs\wmiprov.log**. Errors, such as a bad or missing MOF resource for the driver, are logged here. In the case of a bad MOF resource, the file **%windir%\system32\mofcomp.log** might have additional information related to the error.

You can change the logging settings for all WMI providers by using the **Wmimgmt.msc** application. You can disable or reenable logging, change the directory where WMI log files are kept, as well as set the maximum size for such files.

11.9 WMI Event Tracing

This discussion describes the WMI extensions to WDM that Kernel Mode Drivers, as information providers, can use to provide information to information consumers. Drivers typically provide information that a consumer uses to determine the driver's configuration and resource usage. In addition to the WMI extensions to WDM, a user mode API supports providers or consumers of WMI event information.

The event tracing logger supports up to 32 instances. One of the instances is reserved for tracing the kernel. The logger supports tracing a high event rate.

Trace events are defined in the same manner as other WMI events. WMI events are described in the MOF file.

The process by which Kernel Mode Drivers log information is integrated into the existing WMI infrastructure. To log trace events, a driver does the following:

1. Register as a WMI provider by calling **IoWMIRegistration Control**.
2. Mark events as traceable by setting **WMIREG_FLAG_TRACED_ GUID** in the **Flags** member of the **WMIREGGUID** structure that is passed when the driver registers events with WMI.
3. Specify one event as the control event for overall enabling/disabling of a set of trace events by setting **WMIREG_FLAG_ TRACE_CONTROL_GUID** in the **Flags** member of the **WMIREGGUID** structure that is passed when the driver registers events with WMI.
4. Upon receiving a request from WMI to enable events where the GUID matches the trace control GUID, the driver should store the handle to the logger. The value will be needed when writing an event. For information about how to use this handle, see step 6. The logger handle value is contained in the **HistoricalContext** member of the **WNODE_HEADER** portion of the WMI buffer that is part of the parameters in the enable events request.
5. Decide whether the trace event will be sent to WMI event consumers or is targeted for the WMI event logger only. This determines where the memory for the **EVENT_TRACE_HEADER** structure should come from. This memory eventually is passed to **IoWMIWriteEvent**.

   If the event is a log event only, the memory is not deleted by WMI. In this case, the driver should pass in a buffer on the stack or should be reusing an allocated buffer for this purpose. For performance reasons, the driver should minimize any unnecessary calls to allocate or free memory. Failure to comply with this recommendation compromises the integrity of the timing information contained in the log file.

   If the event is to be sent to both the logger and to WMI event consumers, the memory must be allocated from a nonpaged pool. In this case, the event is sent to the logger and then forwarded to WMI to be sent to WMI event consumers who have requested notification of the event. The memory for the event is then freed by WMI according to the behavior of **IoWMIWriteEvent**.

6. After the memory for the **EVENT_TRACE_HEADER** and any driver event data, if any, has been secured, the following information should be set:
   - Set the **Size** member to the size of (**EVENT_TRACE_HEADER**) plus the size of any additional driver event data that will be appended on to the end of **EVENT_TRACE_HEADER**.
   - Set the **Flags** member to **WNODE_FLAG_TRACED_GUID** to have the event sent to the logger. If the event is to be sent to WMI event consumers as well, set the **WNODE_FLAG_LOG_WNODE**. Note, it is not necessary to set **WNODE_FLAG_TRACED_GUID** if setting **WNODE_FLAG_LOG_WNODE**. If both are set, **WNODE_FLAG_TRACED_GUID** takes precedence and the event is not sent to WMI event consumers.
   - Set the **Guid** or the **GuidPtr** member. If using **GuidPtr**, set **WNODE_FLAG_USE_GUID_PTR** in the **Flags** member. Optionally, specify a value for **TimeStamp**. If the driver does not specify a **TimeStamp** value, the logger fills this in. If the driver does not want the logger to set the time stamp, it should set **WNODE_FLAG_USE_TIMESTAMP** in the **Flags** member.
   - Set any of the following **EVENT_TRACE_HEADER** members that have meaning to the driver—that is, **Class.Type**, **Class.Level**, and **Class.Version**.
   - Finally, cast the **EVENT_TRACE_HEADER** to a **WNODE_HEADER** and set the **HistoricalContext** value of the **Wnode** to the logger handle that was saved in the preceding step 4.
7. Call **IoWMIWriteEvent** with the pointer to the **EVENT_TRACE_HEADER** structure.

The driver should continue logging trace events associated with the control GUID until the driver receives notification to disable event logging via an **IRP_MN_DISABLE_EVENTS** request.

This page intentionally left blank

CHAPTER 12

PROGRAMMING KMDF HARDWARE DRIVER

This chapter will cover the sample **PCIDRV** sample driver, and how to use it to create a KMDF hardware driver.

The sample **PCIDRV** driver is a fully functional KMDF driver for the Intel 82557/82558 based-PCI Ethernet Adapter (10/100) and Intel compatibles. This device can use either port or memory resources to control the device. It can be stopped and started at run time and supports a low-power state. The driver supports multiple concurrent read and write requests for the device.

The **PCIDRV** sample is installed in the samples directory at **pcidrv\sys**. The sample supports the following features in addition to those previously described for the **Toaster** samples:

- Handling Plug and Play and power events
- Supporting DMA interfaces
- Performing event tracing
- Handling interrupts
- Mapping resources
- Using multiple I/O queues and performing device I/O
- Reading and writing to the registry
- Using self-managed I/O to implement a watchdog timer

The **PCIDRV** sample supports physical hardware and contains many lines of device-specific code, so it is significantly more complicated than the **Toaster** drivers.

The sample also uses event tracing for Windows to record driver data for debugging and logging purposes. Event tracing is supported for all Windows drivers.

12.1 Support Device Interrupts

To support interrupt handling for its device, a KMDF driver must

- Create an interrupt object.
- Enable and disable the interrupt.
- Optionally perform pre-enable and post-disable processing that is related to the interrupt.
- Handle interrupts as they occur.

12.1.1 Creating an Interrupt Object

A driver typically creates its interrupt objects (**WDFINTERRUPT**) in its **EvtDriverDeviceAdd** callback. The driver must have an interrupt object for each interrupt vector or message-signaled interrupt (MSI) that each of its devices support. Each interrupt object must include pointers to **EvtInterruptIsr** and **EvtInterruptDpc** event callback functions and may also include additional information.

The framework calls **EvtInterruptIsr** when an interrupt occurs. This callback runs at device interrupt request level (**DIRQL**) for the device and is the equivalent of the WDM **InterruptService** function. The **EvtInterruptIsr** callback queues a **DPC** to perform additional interrupt-related processing. The framework calls the driver's **EvtInterruptDpc** callback when the **DPC** is ready to run. The **EvtInterruptDpc** callback runs at **DISPATCH_LEVEL** and is equivalent of the WDM **DpcForisr** function.

Creating an interrupt object, like creating any other WDF object, involves filling in a configuration structure and calling a creation method. The driver calls the **WDF_INTERRUPT_CONFIG_INIT** function to initialize the **WDF_INTERRUPT_CONFIG** structure with pointers to the **EvtInterruptIsr** and **EvtInterruptDpc** callbacks. After initializing the structure, the driver can set additional information in it, including a pointer to a spin lock and pointers to the **EvtInterruptEnable** and **EvtInterruptDisable** callbacks, which enable and disable interrupts for the device. KMDF calls these functions at **DIRQL** while holding the interrupt spin lock during device power state transitions and when the driver calls **WdfEnableInterrupt** or **WdfDisableInterrupt**.

If the driver must perform additional tasks immediately after the interrupt is enabled and before it is disabled, it should also register the **EvtDeviceDOEntryPostInterruptsEnabled** and **EvtDevice-DOExitPreInterruptsDisabled** callbacks. KMDF calls both of these functions at **PASSIVE_LEVEL** without holding the interrupt spin lock.

To create the interrupt object, the driver calls the **WdfInterrupt-Create** method and passes a handle to the device object, a pointer to the interrupt configuration structure, a pointer to an attribute configuration block, and a pointer to a variable that receives the handle to the interrupt object. Drivers typically specify **WDF_NO_ATTRIBUTES** when creating an interrupt object.

12.1.2 Code to Create an Interrupt Object

The **PCIDRV** sample creates its interrupt object in the **NICAllocate-SoftwareResources** function, which is called by **PciDrvEvtDeviceAdd** (the driver's **EvtDriverDeviceAdd** callback). The following code shows how the **PCIDRV** sample creates its interrupt object:

```
WDF_INTERRUPT_CONFIG_INIT(&interruptConfig,
               NICEvtInterruptIsr,
               NICEvtInterruptDpc);

interruptConfig.EvtInterruptEnable  =
   NICEvtInterruptEnable;
interruptConfig.EvtInterruptDisable =
   NICEvtInterruptDisable;

status = WdfInterruptCreate(FdoData->WdfDevice,
         &interruptConfig,
         WDF_NO_OBJECT_ATTRIBUTES,
         &FdoData->WdfInterrupt);

If(!NT_SUCCESS (status)) {
   return status;
}
```

The **PCIDRV** sample configures the interrupt object by specifying pointers to **NICEvtInterruptIsr** and **NICEvtInterruptDpc**, which are called to handle interrupts and to perform deferred interrupt processing at **IRQ DISPATCH_LEVEL**, respectively. The driver also sets pointers to **NICEvtInterruptEnable** and **NICEvtInterruptDisable**, which enable and disable interrupts in the hardware. The call to **WdfInterruptCreate** returns a handle to the interrupt object, which the driver stores in the context area of its device object.

12.1.3 Enabling and Disabling Interrupts

A KMDF driver's **EvtInterruptEnable** callback enables interrupts for its device. KMDF calls this function during a device transition to **DO**, after **EvtDeviceDOEntry** has returned. The callback is called at **DIRQL** for the device with the interrupt spin lock held; therefore, the callback should quickly enable the interrupt and return. If the driver requires additional processing after enabling its interrupt, it should set an **EvtDeviceDO-EntryPostInterruptsEnable** callback, which KMDF calls at **PASSIVE_LEVEL**.

EvtInterruptEnable is called with two parameters: a handle to the interrupt object and a handle to the device object that is associated with the interrupt object. The driver passes the device object handle to the accessor function for its device context area, where it has stored information about its device registers. With the returned pointer, the driver can access the hardware registers as required to enable the interrupt.

A KMDF driver's **EvtInterruptDisable** callback disables interrupts for its device. KMDF calls this function during a device transition out of the **DO** state, but before it calls **EvtDeviceDOExit**. Like **EvtInterrupt-Enable**, **EvtInterruptDisable** is called at **DIRQL** for the device and with the interrupt spin lock held; therefore, it should quickly disable the interrupt and return. If the driver requires additional processing before disabling its interrupt, it should set an **EvtDeviceDOExit-PreInterruptsDisabled** callback, which KMDF calls at **PASSIVE_LEVEL**.

The **EvtInterruptDisable** callback is passed the same two parameters as the **EvtInterruptEnable** callback and proceeds to undo the actions that were performed in that callback.

12.1.4 Code to Enable Interrupts

In the **PCIDRV** sample, the **EvtInterruptEnable** callback appears in **isrdpc.c**. The required code is quite simple, as the following shows:

```
NTSTATUS
NICEvtInterruptEnable(
    IN    WDFINTERRUPT    Interrupt,
    IN    WDFDEVICE       AssociateDevice
    )
{
    PFDO_DATA       fdoData;
    fdoData = FdoGetData(AssociateDevice);
    NICEnalbeInterrupt(Interrupt, fdoData);
    return STATUS_SUCCESS;
}
```

NICEnableInterrupt is simply a macro that sets the PCI device register to enable interrupts.

12.1.5 Code to Disable Interrupts

In the **PCIDRV** sample, the **EvtInterruptDisable** callback (also in **isrdpc.c**) simply calls the internal **NICDisableInterrupt** macro to set the PCI device register to disable interrupts. The following is the code:

```
NTSTATUS
NICEvtInterruptDisable(
    IN    WDFINTERRUPT    Interrupt,
    IN    WDFDEVICE       AssociatedDevice
    )
{
    PFDO_DATA    fdoData;

    fdoData = FdoGetData(AssociatedDevice);
    NICDisableInterrupt(fdoData);
    return STATUS_SUCCESS;
}
```

12.1.6 Post-Interrupt Enable and Pre-Interrupt Disable Processing

Some devices cause interrupt storms if they are initialized after their interrupts are enabled. The driver for such a device must therefore be able to perform initialization after the device enters **DO** but before its interrupt is enabled. Other devices, however, cannot be initialized until the interrupt is enabled. To enable correct operation of both types of devices, KMDF supplies post-interrupt enable and pre-interrupt disable events for which drivers can register.

When powering up a device, KMDF invokes a driver's callbacks in the following order:

1. **EvtDeviceDOEntry**
2. **EvtInterruptEnable**
3. **EvtDeviceDOEntryPostInterruptsEnabled**

KMDF calls **EvtDeviceDOEntry** first. Drivers that must initialize their devices before the interrupt is connected (for example, to prevent interrupt storms) should do so in this callback. Next, KMDF calls **EvtInterruptEnable**. Drivers should enable their interrupts and do little or nothing else in this function because it is called at **DIRQL**. Finally, KMDF calls **EvtDeviceDOEntryPostInterruptsEnabled** at **PASSIVE_LEVEL**. Drivers that must initialize their devices after the interrupt is connected should do so in this callback.

At power-down, KMDF calls the corresponding paired functions in the opposite order:

1. **EvtDeviceDOExitPreInterruptsDisabled**
2. **EvtInterruptDisable**
3. **EvtDeviceDOEXit**

To undo work done by the **EvtDeviceDOEntryPostInterrupts-Enabled**, a driver registers an **EvtDeviceDOExitPreInterrupts-Disabled** function. Like the post-enable function, the pre-disable function does work at **PASSIVE_LEVEL** in preparation for disabling the interrupt.

A driver must register the post-interrupt enable and pre-interrupt disable callbacks in the **WDF_PNPPOWER_EVENT_CALLBACKS** structure before creating the device object. The **PCIDRV** sample fills this structure in its **EvtDriverDeviceAdd** callback, as follows:

```
pnpPowerCallbacks.EvtDeviceDOEntryPostInterruptsEnabled =
      NICEvtDeviceDOEntryPostInterruptsEnabled;
pnpPowerCallbacks.EvtDeviceDOExitPreInterruptsDisabled =
      NICEvtDeviceDOExitPreInterruptsDisabled;
```

In the current version of the **PCIDRV** sample, both of these functions are stubs.

12.2 Handling Interrupts

When a device interrupts, Windows calls the driver to service the interrupt. However, more than one device can be connected to a single interrupt vector. Internally, the operating system keeps a list of the **InterruptService** routines of the drivers for devices that interrupt at the same level. When an interrupt signal arrives, Windows traverses the list and calls the drivers in sequence until one of them acknowledges and services the interrupt.

KMDF intercepts the call from the operating system and, in turn, calls the **EvtInterruptIsr** callback that the driver registered. Like traditional **InterruptService** functions, this callback runs at **DIRQL**. Because **EvtInterruptIsr** runs at such a high **IRQL**, it should do its work quickly and then return. Remember that it cannot take any action that would cause a page fault (or wait on any dispatcher objects) because it runs at an **IRQL** greater than **APC_LEVEL**.

The **EvtInterruptIsr** callback should perform the following tasks, and nothing more:

- Determine whether its device is interrupting, and if not, return **FALSE** immediately.
- Stop the device from interrupting.
- Queue a **DPC** to perform any work related to the interrupt.

The **EvtInterruptIsr** function is called with a handle to the interrupt object for the driver's device and a **ULONG** value that specifies the message **ID** if the device is configured for MSIs and zero otherwise.

12.2.1 Code for EvtInterruptIsr Callback

The following is the **PCIDRV** sample's **EvtInterruptIsr** callback (which is defined in **pcidrv\sys\hw\isrdpc.c**):

```
BOOLEAN
NICEvtInterruptIsr(
    IN    WDFINTERRUPT    Interrupt,
    IN    ULONG           MessageID
    )
{
    BOOLEAN        InterruptRecognized = FALSE;
    PFDO_DATA      FdoData = NULL;
    USHORT         IntStatus;

    UNREFERENCED_PARAMETER (MessageID);

    FdoData = FdoGetData (WdfInterruptGetDevice (Interrupt));

    //
    // Process the interrupt if it is enabled
    // and active.
    //
    If (!NIC_INTERRUPT_DISABLED (FdoData)  &&
       NIC_INTERRUPT_ACTIVE (FdoData))
    {
       InterruptRecognized = TRUE;

       //
       // Disable the interrupt. It will be reenabled in
       // NICEvtInterruptDpc.
       //

       NICDisableInterrupt (FdoData);

       //
       // Acknowledge the interrupt (s) and get status
       //

       NIC_ACK_INTERRUPT (FdoData, IntStatus);
       WdfInterruptQueueDpcForIsr (Interrupt);
    }
    return InterruptRecognized;
}
```

The **PCIDRV** sample's first step is to determine whether its device is interrupting. To do so, it must check its device registers. It gets a handle to the device object that is associated with the interrupt object by calling the **WdfInterruptGetDevice** method and then passes that handle to **FdoGetData** to get a pointer to the device context area. In the context area, the driver stored a pointer to its mapped hardware registers.

The **NIC_INTERRUPT_DISABLED** and **NIC_INTERRUPT_ACTIVE** macros (defined in **pcidrv\sys\Hw\Nic_def.h**) check the hardware registers to determine whether interrupts have been disabled for the device and whether they are currently active. If interrupts have been disabled, the device cannot possibly have generated the interrupt. The same is true if the device's interrupt is enabled but not currently active. In either case, the driver returns with **InterruptRecognized** set to **FALSE**. (For most drivers, checking whether device interrupts have been disabled is unnecessary.)

However, if interrupts have not been disabled and an interrupt is currently active, the device must have generated the interrupts. In this case, the driver sets **InterruptRecognized** to **TRUE**.

To stop the device from interrupting, the driver calls **NICDisable-Interrupt** and then uses the driver-defined **NIC_ACK_INTERRUPT** macro to acknowledge the interrupt in the hardware. Finally, it queues a **DPC** by calling **WdfInterruptQueueDpcForIsr** and then returns.

12.2.2 Deferred Processing for Interrupts

When the **DPC** runs, KMDF calls the driver's **EvtInterruptDpc** callback. This function performs device-specific interrupt processing and reenables the interrupt for the device. It runs at **DISPATCH_LEVEL** and therefore must not attempt any operations that might cause a page fault (or wait on any dispatcher objects).

12.2.2.1 Code for EvtInterruptDpc Callback

The following is the **PCIDRV** sample's **EvtInterruptDpc** callback:

```
VOID
NICEvtInterruptDpc(
    IN    WDFINTERRUPT    WdfInterrupt,
    IN    WDFOBJECT        WdfDevice
    )
{
    PFDO_DATA   fdoData = NULL;

        fdoData = FdoGetData (WdfDevice);

        WdfSpinLockAcquire (fdoData->RcvLock);
        NICHandleRecvInterrupt (fdoData);
        WdfSpinLockRelease (fdoData->RcvLock);

        //
        // Handle send interrupt.
        //

        WdfSpinLockAcquire (fdoData->SendLock);
        NICHandleSendInterrupt (fdoData);
        WdfSpinLockRelease (fdoData->SendLock);

        //
        // Check if any queued sends need to be reprocessed.
        //
        NICCheckForQueuedSends (fdoData);

        //
        // Start the receive unit if it was stopped
        //

        WdfSpinLockAcquire (fdoData->RcvLock);
        NICStartRecv (fdoData);
        WdfSpinLockRelease (fdoData->RcvLock);

        //
        // Reenable the interrupt.
        //

        WdfInterruptSynchronize (
           WdfInterrupt,
           NICEnableInterrupt,
           fdoData);

    return;
    }
```

Most of the code in this callback is device specific. However, its use of spin locks is worth noting.

When the driver created its I/O queues, it also created two spin locks and stored their handles in its device context area. One (**RcvLock**) protects read-related buffers and operations, and the (**SendLock**) protects write-related buffers and operations. In this function, it uses these spin locks to protect against preemption and concurrent users while it processes the results of the I/O operation. The driver calls **WdfSpinLockAcquire** and **WdfSpinLockRelease** to acquire and release the locks.

When the driver has completed all device-specific processing, it reenables the interrupt. The function that reenables the interrupt (**NICEnableInterrupt**) must run at **DIRQL**, so the driver calls **WdfInterruptSynchronize** to run it. **WdfInterruptSynchronize** takes a handle to the interrupt object, a pointer to the function to be run (**NICEnableInterrupt**), and a pointer to the device context area, which it passed as an argument to **NICEnableInterrupt**. **WdfInterrupt-Synchronize** raises **IRQL** to **DIRQL** and calls **NICEnableInterrupt**. When **NICEnableInterrupt** completes, KMDF lowers the **IRQL** to **DISPATCH_LEVEL** and returns.

12.3 Mapping Resources

A KMDF driver maps resources for its hardware as part of its **EvtDevicePrepareHardware** callback and unmaps them in its **EvtDeviceReleaseHardware** callback. These callbacks provide a way for a driver to prepare its device hardware immediately before the device enters and after the device leaves the operational (**DO**) state. These two routines are always called in pairs—that is, after KMDF calls **EvtDevicePrepareHardware**, it always calls **EvtDeviceRelease-Hardware** before calling **EvtDevicePrepareHardware** again.

KMDF calls a driver's **EvtDevicePrepareHardware** callback before calling **EvtDeviceDOEntry** whenever resources are assigned to the device—specifically, during initial device enumeration and during power-up after resource rebalancing. **EvtDevicePrepareHardware** should map device resources but should not load firmware or perform other device initialization tasks. Drivers for USB devices might also get device and configuration descriptors and select configurations in this call-back. Drivers should not attempt to access their device hardware in this callback.

EvtDeviceReleaseHardware undoes any work that was done by **EvtDevicePrepareHardware**. For example, if **EvtDevicePrepare-Hardware** maps resources, **EvtDeviceReleaseHardware** releases them.

Within its **EvtDevicePrepareHardware** function, a driver calls **WdfCmResourceListGetCount** to get the number of resources that the system has assigned to its device and then calls **WdfCmResourceList-GetDescriptor** to get details about a particular resource from the list.

The chipset on an individual machine can map hardware resources into either I/O or memory space, regardless of how the device itself is designed. To be platform independent, all drivers should support both types of mapping, just as the **PCIDRV** sample does. For I/O and memory-mapped resources, a KMDF driver takes steps that are similar to those a WDM driver would take:

- For an I/O-mapped resource (**CmResourceTypePort**), the driver saves the base address and range at which the resource is mapped and saves a pointer to the HAL's ***_PORT_*** functions that it will use to access the resource.

- For a memory-mapped resource (**CmResourceTypeMemory**), the driver checks that the allocated size is adequate. If so, it maps the returned physical address to a virtual address by calling MmMapIoSpace and saves pointers to the HAL's ***_REGISTER_*** functions that it will use to access the resource.

For interrupt resources, a KMDF driver simply creates an interrupt object as part of **EvtDriverDeviceAdd** processing, as described in the previous discussion. The object itself picks up its resources with no required driver intervention.

12.3.1 Code to Map Resources

The PCI device supported by the sample driver has three base address registers (**BARs**): **BAR 0** is memory mapped, **BAR 1** is I/O mapped, and **BAR 3** is flash-memory mapped. The driver determines whether to use the I/O-mapped **BAR** or the memory-mapped **BAR** to access the control and status registers.

The sample driver checks for registers in both memory and I/O space. On some platforms, the I/O registers can be mapped into memory space; every driver should be coded to handle this.

In the **PDIDRV** sample, the code to map resources is isolated in the **NICMapHwResources** function, which is called by **PciDrvEvtDevice-PrepareHardware**. **NICMapHwResources** has two parameters: a pointer to the device context area (**FdoData**) and a handle to the list of translated resources (**ResourcesTranslated**) that was passed to **PciDrvEvtDevicePrepareHardware**. The driver must use the translated resources to map device registers into port and memory space.

The following sample code is excerpted from the **pcidrv\sys\Hw\Nic_init.c** file. It shows how the **PCIDRV** sample maps hardware resources.

```
NTSTATUS
NICMapHwResources (
   IN   OUT   PFDO_DATA   FdoData,
   WDFCMRESLIST       ResourcesTranslated
   )
{
   PCM_PARTIAL_RESOURCE_DESCRIPTOR       descriptor;
   ULONG       i;
   NTSTATUS    status                  = STATUS_SUCCESS;
   BOOLEAN     bResPort                = FALSE;
```

```
    BOOLEAN     bResInterrupt         = FALSE;
    BOOLEAN     bResMemory            = FALSE;
    ULONG       numberOfBARSs         = 0;

    PAGED_CODE ();

for  (i = 0; i<WdfCmResourceListGetCount
     (ResourcesTranslated); i++)
    {
        descriptor =
            WdfCmResourceListGetDescriptor
               (RessourcesTranslated, i);

            If (!descriptor)
            {
                return STATUS_DEVICE_CONFIGURATION_ERROR;
            }

            switch (descriptor->Type)
            {
            case CmResosurceTypePort:
                //
                // We will increment the BAR count only for valid
                // resources. Do not count the private device types
                // added by the PCI bus driver.
                //

                numberOfBars++;

//
// The resources are listed in the same order as the
// BARSs in the configuration space, so this should
// be the second one.
//

if (numberOfBars != 2)
{
status = STATUS_DEVICE_CONFIGURATION_ERROR;
return status;
}
//
// The port is in I/O space on this machine.
// We should use READ_PORT_Xxx
// and WRITE_PORT_Xxx functions to read or
// write to the port.
//
```

```
FdoData->IoBaseAddress =
  ULongToPtr (descriptor->u.Port.Start.LowPart);
FdoData->IoRange = descriptor->u.Port.Length;

//
// Since all our accesses are USHORT wide, we will
// create an accessor table just for these two
// functions.
//

FdoData->ReadPort = NICReadPortUShort;
FdoData->WritePort = NICWritePortUShort;

bResPort = TRUE;
FdoData->MappedPorts = FALSE;
break;

          case CmResourceTypeMemory:
             numberOfBARs++;
             if (numberOfBARs  == 1)
             {
                //
                // Our CSR memory space should be 0x1000 in
                // size.
                //
ASSERT (descriptor->u.Memory.Length == 0x1000);
FdoData->MemPhysAddress =
   descriptor->u.Memory.Start;
FdoData->CSRAddress = MmMapIoSpace (
  descriptor->u.Memory.Start,
  NIC_MAP_IOSPACE_LEGTH,
  MmNonCached);
                bResMemory = TRUE;
          }
             else if (numberOfBARs == 2)
             {
                //
               // The port is in memory space on this machine.
               // Call MmMapIoSpace to map the
               // physical to virtual address, and use the
               // READ/WRITE_REGISTER_xxx function
               // to read or write to the port.
               //
                FdoData->IoBaseAddress = MmMapIoSpace (
                  descriptor->u.Memory.Start,
                  descriptor->u.Memory.Length,
                  MmNonCached);
                FdoData->ReadPort = NICReadRegisterUShort;
                FdoData->WritePort = NICWriteRegisterUShort;
                FdoData->MappedPorts = TRUE;
                bResPort = TRUE;
            }
            else if (numberOfBARs == 3)
            {
               //
               // Our flash memory should be 1MB in size. We
               // don't access it, so do not bother mapping it.
               //
               // ASSERT (descriptor->u.Memory.Length ==
               //          0x100000);
            }
            else
            {
                status =
                    STATUS_DEVICE_CONFIGURATION_ERROR;
                    return status;
            }
            break;

        case CmResourceTypeInterrupt:
            ASSERT (!bResInterrupt);
            bResInterrupt = TRUE;
            break;

        default:
            //
            // This could be a device-private type added by
            // the PCI bus driver. We shouldn't filter this
            // or change the information contained in it.
            //
            break;
        }
    }


    // Make sure we got all the resources to work with.
    //

    if (! (bResPort && bResInterrupt && bResMemory))
    {
        status =
        STATUS_DEVICE_CONFIGURATION_ERROR;
    }
    return status;
}
```

The driver parses the list of translated resources in a loop that starts at zero and ends when it has reached the last resource in the list. The driver determines the number of resources in the list by calling the **WdfCmResourceListGetCount** function.

For each resource in the list, the driver calls **WdfCmResourceListGetDescriptor** to get a pointer to the resource descriptor. The resource descriptor is an enumerator that indicates the type of the resource. (If you are familiar with WDM drivers, you will notice that the resource types are the same as those for WDM.)

For the **CmResourceTypePort** resources, the driver saves the starting address and range in the device context area and then sets the addresses of the functions that it uses to access the port resources.

For **CmResourceTypeMemory** resources, the driver also saves the starting address and range in the device context area, but then uses **MmMapIoSpace** to map the resources and get a virtual address through which it can access them.

For **CmResourceTypeInterrupt** resources, the driver is not required to save the resource information because KMDF handles this transparently for the driver when the driver creates the **WDFINTERRUPT** object. The sample driver merely checks this resource for completeness.

12.3.2 Code to Unmap Resources

When the device is removed or when the system rebalances resources, the driver must release its mapped resources in an **EvtDeviceReleaseHardware** callback. KMDF calls this function after calling the driver **EvtDeviceDOExit** function.

The **PCIDRV** sample does so in the internal function **NICUnmapHwResources**, which is called by its **EvtDeviceReleaseHardware**. **NICUnmapHwResources** appears in the **pcidrv\sys\hw\nic_init.c** source file and releases the resources as follows:

```
if (FdoData->CSRAddress)
{
   MmUnmapIoSpace (FdoData->CSRAddress,
         NIC_MAP_IOSPACE_LENGTH);
```

```
        FdoData->CSRAddress = NULL;

    }
    if (FdoData->MappedPorts)
    {
        MmUnMapIoSpace (FdoData->IoBaseAddress,
                FdoData->IoRange);
        FdoData->IoBaseAddress = NULL;
    }
```

This page intentionally left blank

CHAPTER 13

Programming Multiple I/O Queues and Programming I/O

This chapter covers the important aspect of device driver development involving the handling of I/O queues.

13.1 Introduction to Programming I/O Queues

Drivers have several options when creating and configuring I/O queues. The most important are which I/O requests to queue, what type of dispatching to use, and whether to let KMDF manage the queue. A device object can have any number of I/O queues, and each can be configured differently.

One queue for each device object can be configured as a default queue, into which KMDF places requests for which the driver has not specifically configured any other queue. If the device object has a default queue and one or more other queues, KMDF queues specific requests to the correspondingly configured queues and queues all other requests to the default queue. If the device object does not have a default queue, KMDF queues only the specific request types to the configured queues and, for a function or bus driver, fails all other requests. (For a filter driver, it passes all other requests to the next lower driver.)

A queue's dispatch type determines when KMDF presents I/O requests from the queue to the driver. KMDF supports three dispatch types:

- Sequential
- Parallel
- Manual

KMDF presents requests from a sequential queue to the driver one at a time. When the first request arrives, KMDF calls the event callback that was specified for the request type and queue. After the driver completes the request, KMDF calls the event callback for another request if the queue is not empty. A sequential queue is thus used for synchronous I/O. By default, queues are configured as sequential (**WdfIoQueueDispatch-Sequential**).

For a parallel queue, KMDF presents requests to the driver as soon as they arrive. KMDF does not wait for any "inflight" requests to complete before sending another request. An **inflight** request is an I/O request that is currently active—that is, it is not in a queue and has not been completed. A parallel queue can thus be used for asynchronous I/O.

For a manual queue, KMDF does not present requests to the driver. Instead, the driver must call **WdfIoQueueRetrieveRequest** when it is ready to handle a request. A driver can thus use manual queuing for either synchronous or asynchronous I/O.

A driver can temporarily stop the delivery of requests from a sequential or parallel queue by calling **WdfIoQueueStop** or **WdfIoQueue-StopSynchronously**, depending on the type of queue and the reasons for pausing delivery. To restart the queue, the driver calls **WdfIoQueueStart**.

By default, KMDF handles I/O cancellation for queued I/O requests. Consequently, if the user cancels an I/O request after KMDF has queued it but before KMDF has delivered it to the driver, KMDF removes it from the queue and completes it with **STATUS_CANCELED**. The driver can request notification by registering an **EvtIoCanceledOnQueue** callback for the queue; otherwise, KMDF cancels the request without notifying the driver.

After a request has been dispatched to the driver, it cannot be canceled unless the driver specifically marks it as cancelable and registers an **EvtRequestCancel** callback. If the driver forwards the request to another queue, it immediately becomes cancelable again.

Also by default, KMDF handles power management for I/O queues, and each I/O queue inherits the power state of its associated device. During Plug and Play or power state transitions and any time the device is not in the working state, KMDF queues incoming I/O requests but does not dispatch them to the driver. Therefore, if the driver creates its queues before the device enters **DO**, the queues are in the **WDF_IO_QUEUE_STOPPED** state, and KMDF queues any I/O requests targeted at the device. When the device enters the working state, KMDF resumes presenting requests. A driver can change this default when it configures each queue by setting the **PowerManaged** field of the configuration structure to **FALSE**.

A driver can also specify whether a queue accepts I/O requests that have a zero-length buffer. By default, KMDF does not dispatch such requests to the driver; instead, it completes them with **STATUS_SUCCESS**.

13.2 Creating and Configuring the Queues

To create and configure a queue, a driver takes the following steps:

1. Defines a **WDF_IO_QUEUE_CONFIG** structure to hold configuration settings for the queue.
2. Initializes the configuration structure by calling the **WDF_IO_QUEUE_CONFIG_INIT** or **WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE** function (for a default queue). These functions take a pointer to the configuration structure and an enumerator that defines the dispatching type for the queue.
3. Sets the event callbacks for this queue in the **WDF_IO_QUEUE_CONFIG** structure, if the queue uses sequential or parallel dispatching. A driver can set callbacks for one or more of the following I/O events: **EvtIoRead, EvtIoWrite, EvtIoDeviceIoControl, EvtIoInternalDeviceIoControl, EvtIoDefault, EvtIoStop, EvtIoResume**, and **EvtIoCanceledOnQueue**.
4. Sets Boolean values for the **PowerManaged** and **AllowZeroLengthRequests** fields in the queue configuration structure if the default values are not suitable.
5. Creates the queue by calling **WdfIoQueueCreate**, passing a handle to the **WDFDEVICE** object, a pointer to the filled-in **WDF_IO_QUEUE_CONFIG** structure, a pointer to a **WDF_OBJECT_ATTRIBUTES** structure, and a pointer to receive a handle to the created queue instance.
6. Specifies which I/O requests KMDF should direct to the queue by calling **WdfDeviceConfigureRequestDispatching**.

The **PCIDRV** sample creates two parallel queues and three manual queues, which are used as follows:

- For write requests, the driver creates a parallel queue. If a write request cannot be satisfied immediately, the driver puts the request into an internal manual queue. (The queue is considered internal because only the driver, and not KMDF, adds requests to it.)
- For read requests, the driver creates a manual queue.
- For **IOCTL** requests, the driver creates a parallel queue. If the **IOCTL** cannot be satisfied immediately, the driver puts the request into an internal manual queue.

In this driver, each I/O queue is configured for a particular type of I/O request. Therefore, the driver does not create a default queue.

The code to create all the queues is excerpted from the **NICAllocateSoftwareResources** function (in the file **cidrv\sys\hw\Nic_init.c**), which is called from the driver's **EvtDriverDeviceAdd** callback.

13.2.1 Code to Create Queues for Write Requests

The following excerpt shows how the **PCIDRV** sample creates a parallel queue for incoming write requests. While requests are in this queue, KMDF handles cancellation without notifying the driver.

```
NTSTATUS                  status;
WDF_IO_QUEUE_CONFIG       ioQueueConfig;
WDF_OBJECT_ATTRIBUTES    attributes;

......

   WDF_IO_QUEUE_CONFIG_INIT (
      &ioQueueConfig,
      WdfIoQueueDispatchParallel
   );

   ioQueueConfig.EvtIoWrite = PciDrvEvtIoWrite;

   status = WdfIoQueueCreate (
               FdoData->WdfDevice,
               &ioQueueConfig,
               WDF_NO_OBJECT_ATTRIBUTES,
                &FdoData->WriteQueue    // queue handle
             );

   if (!NT_SUCCESS (status))
   {
      return status;
   }

   status = WdfDeviceconfigureRequestDispatching (
               FdoData->WdfDevice,
               FdoData->WriteQueue,
               WdfRequestTypeWrite );

if (!NT_SUCCESS (status))
   {
      ASSERT (NT_SUCCESS (status));
      return status;
   }
```

The driver calls **WDF_IO_QUEUE_CONFIG_INIT** to initialize the queue as a parallel queue. The queue holds only write requests, so the driver sets only an **EvtIoWrite** callback in the **ioQueueConfig** structure. It creates the queue, and then calls **WdfDeviceConfigureRequest-Dispatching** to configure the queue for requests of type **WdfRequestTypeWrite** only. All other I/O requests are directed to a queue that is configured for them or, if the driver has not configured a queue for them, are handled by the framework without being sent to the driver.

The **PCIDRV** sample also creates a manual queue into which to place pending write requests. The driver's **EvtIoWrite** callback places an I/O request in this internal queue when it cannot handle the request immediately.

The driver creates this queue as follows:

```
WDF_IO_QUEUE_CONFIG_INIT (
   &ioQueueConfig,
   WdfIoQueueDispatchManual
   );

status = WdfIoQueueCreate (
   FdoData->WdfDevice,
   &ioQueueConfig,
   WDF_NO_OBJECT_ATTRIBUTES,
   &FdoData->PndingWriteQueue
   );
```

```
if (!NT_SUCCESS (status))
{
   return status;
}
```

To create the manual internal queue for its pending write requests, the driver configures a **WDF_IO_QUEUE_CONFIG** structure, this time setting the queue type to **WdfIoDispatchManual**, which indicates that the driver requests items from the queue when it is ready. It then creates the queue by calling **WdfIoQueueCreate**. The driver does not configure the queue for any particular type of I/O request because the driver itself determines which request to queue. KMDF does not put any request in this queue because the driver did not call **WdfDevice-configureRequestDispatching**. However, KMDF handles power management and request cancellation.

13.2.2 Code to Create Queues for Read Requests

The driver creates a manual queue for read requests in the same way that it did for write requests. The only difference is that it configures the queue for request dispatching, so that KMDF places read requests directly into the queue. The driver creates the queue as follows:

```
WDF_IO_QUEUE_CONFIG_INIT (
   &ioQueueConfig,
   WdfIoQueueDispatchManual
   );

status = WdfIoQueueCreate (
            FdoData->WdfDevice,
            &ioQueueConfig,
            WDF_NO_OBJECT_ATTRIBUTES,
            &FdoData->PendingReadQueue
        );

if (!NT_SUCCESS (status))
{
   return status;
}

Status = WdfDeviceConfigureRequestDispatching (
            FdoData->WdfDevice,
            FdoData->PendingReadQueue,
            WdfRequesttypeRead
         );
```

```
if (!NT_SUCCESS (status))
{
  ASSERT (NT_SUCCESS (status));
   return status;
}
```

13.2.3 Code to Create Queues for Device I/O Control Requests

The driver creates a second pair of parallel and internal manual queues for **IRP_MJ_DEVICE_IO_CONTROL** requests, just as it did for write requests, as follows:

```
WDF_IO_QUEUE_CONFIG_INIT (
   &ioQueueConfig,
   WdfIoQueueDispatchParallel
   );

ioQueueConfig.EvtIoDeviceControl =
      PciDrvEvtIoDeviceControl;

status = WdfIoQueueCreate (
            FdoData->WdfDevice,
            &ioQueueConfig,
            WDF_NO_OBJECT_ATTRIBUTES,
            &FdoData->IoctlQueue
         );

if (!NT_SUCCESS (status))
{
   return status;
}

status = WdfDeviceConfigureRequestDispatching (
            FdoData->WdfDevice,
            FdoData->IoctlQueue,
            WdfRequestTypeDeviceControl
         );

if (!NT_SUCCESS (status))
```

```
{
   ASSERT (NT_SUCCESS (status));
   return status;
}

// Create internal queue for pending IOCTL requests.

WDF_IO_QUEUE_CONFIG_INIT (
   &ioQueueConfig,
   WdfIoQueueDispatchManual
   );

status = WdfIoQueueCreate (
            FdoData->WdfDevice,
            &ioQueueConfig,
            WDF_NO_OBJECT_ATTRIBUTES,
            &FdoData-PendingIoctlQueue
         );

if (!NT_SUCCESS (status))
{
   return status;
}
```

The driver configures these two queues exactly as it did the parallel and manual queues for write requests. The only difference is that it set a callback for **EvtDeviceIoControl** instead of **EvtIoWrite** and configures request dispatching for **WdfRequestTypeDeviceControl** instead of **WdfRequestTypeWrite**.

13.3 Handling Requests from a Parallel Queue

KMDF delivers I/O requests from a parallel queue by calling the appropriate callback function that the driver registered for the queue. For example, the **PCIDRV** sample configures its parallel device I/O control queue, which accepts only **WdfRequestTypeDeviceControl** requests, with an **EvtIoDeviceControl** callback.

The driver's actions in the callback depend on the type of I/O that is required to satisfy the request. If the request required DMA, the driver should retrieve a pointer to the device context area and then set up the DMA transaction.

For I/O requests that do not involve DMA, a driver takes the following steps:

1. Gets the parameters for the I/O request.
2. Parses the parameters.
3. Performs the requested I/O or manually requeues the request for later processing.

13.3.1 Code to Handle I/O Requests

The following code shows how the **PCIDRV** sample handles certain device I/O control requests. The **PciDrvEvtIoControl** function is the driver's **EvtIoDeviceControl** callback for one of its parallel queues and appears in the **pcidrv\sys\Pcidrv.c** source file.

```
VOID
PciDrvEvtIoDeviceControl (
   IN   WDFQUEUE      Queue,
   IN   WDFREQUEST      Request,
   IN   size_t          OutputBufferLength,
   IN   size_t          InputBufferLength,
   IN   ULONG           IoControlCode
   )
{
   NTSTATUS             status = STATUS_SUCCESS;
   PFDO_DATA            fdoData = NULL;
   WDFDEVICE            hDevice;
   WDF_REQUEST_PARAMETERS      params;

   UNREFERENCED_PARAMETER (OutputBufferLength);
   UNREFERENCED_PARAMETER (InputBufferLength);


   hDevice = WdfIoQueueGetDevice (Queue);
   fdoData = FdoGetData (hDevice);

   WDF_REQUEST_PARAMETERS_INIT (&params);

   WdfRequestGetParameters (
      Request,
      &params
      );
```

```
    switch (IoControlCode)
    {
       case IOCTL_NDISPROT_QUERY_OID_VALUE:
           ASSERT ((IoControlCode * 0x3) ==
              METHOD_BUFFERED);
NICHandleQueryOldRequest (
   Queue,
   Request,
   &params
   );
break;

       // code omitted
       //  …..
       //
          case IOCTL_NDISPROT_INDICATE_STAUS:
             status = WdfRequestForwardToIoQueue (Request,
                        fdoData->PendingIoctlQueue);
             if (!NT_SUCCESS (status))
             {
                WdfRequestcomplete (Request, status);
             break;
             }
          break;

          default:
             ASSERTMSG (FALSE, "Invalid IOCTL request\n");
             WdfRequestComplete (Request,
                  STATUS_INVALID_DEVICE_REQUEST);
             break;
       }
       return;
    }
```

The driver requires access to its device object context area, so it starts by calling **WdfIoQueueGetDevice**, which returns a handle to the device object that is associated with the I/O queue. It then passes the returned handle to **FdoGetData**, the accessor function for its device context area, which returns a pointer to the context area.

Next, the driver retrieves the parameters that were passed with the I/O request. KMDF defines the **WDF_REQUEST_PARAMETERS** structure for this purpose. The driver initializes the structure by calling **WDF_REQUEST_PARAMETERS_INIT** and then passes it to the **WdfRequestParameters** method. This method fills in the requested parameters. The driver must retrieve the parameters from the request structure before it performs the I/O.

The driver's action depends on the I/O control code that was specified in the request. For most control codes, the driver calls a device-specific function to handle the request. If the control code is valid, the driver performs the requested action. If the control code is not valid, the driver completes the request by calling **WdfRequestComplete** with the handle to the request and the status **STATUS_INVALID_DEVICE_REQUEST**.

In response to the two control codes shown in the code, the driver takes two different actions. If the control code is **IOCTL_NDISPROT_QUERY_OID_VALUE**, the driver performs buffered I/O. If the control code is **IOCTL_NDISPROT_INDICATE_STATUS**, the driver forwards the request to a different I/O queue. The next two sections describe these actions.

13.3.2 Performing Buffered I/O

The **WDFREQUEST** object contains pointers to the input and output buffers for the I/O request. The driver can obtain these pointers by calling **WdfRequestRetrieveOutputBuffer** and **WdfRequestRetrieveInputBuffer**.

For **METHOD_BUFFERED** requests, however, the input and output buffers are the same, so both of these methods return the same pointer. Therefore, when performing buffered I/O, a driver must read all input data from the buffer before writing any output data to the buffer.

The **NICHandleQueryOidRequest** function, defined in the **pcidrv\sys\hw\Nic_req.c** file, gets the input and output buffers from the **WDFREQUEST** object and retrieves the requested information. Most of this function performs device-specific tasks, so this section describes only the KMDF-specific actions.

The sample retrieves the buffer by calling **WdfRequestRetrieveOutputBuffer**, as follows:

```
status = WdfRequestRetrieveOutputBuffer (Request,
      sizeof (NDISPROT_QUERY_OID),
      &DataBuffer,
      &BufferLength
      );

if (!NT_SUCCESS (status))
```

```
{
   WdfRequestcomplete (Request, status);
   return;
}
```

The input parameters to **WdfRequestRetrieveOutputBuffer** are the handle to the request object and the minimum required size for the buffer. The method returns a pointer to the buffer and a pointer to its length. If the length is smaller than the minimum required or if some other error occurs, **WdfRequestRetrieveOutputBuffer** returns a failure status and the driver immediately completes the request with that status.

The driver then performs device-specific queries for the requested information and writes the returned data into the output buffer. If the buffer is too small to hold the data, the driver sets status to **STATUS_BUFFER_TOO_SMALL**.

Next, the driver updates the buffer length to include the number of bytes of data, the size of the structure that holds it, and any padding bytes that are required to align the structure correctly. The driver performs this calculation even if the buffer is too small, so that it can return the buffer length that would be required to successfully complete the request. Finally, the driver completes the request by calling **WdfRequest-CompleteWithInformation** with the handle to the request, the previously set status value, and the calculated buffer length, as follows:

```
//
// Adjust the size to include the structure.
//
ulInfoLen += FIELD_OFFSET (NDISPROT_QUERY_OID, Data);

WdfRequestCompleteWithInformation (Request,
      Status, ulInfoLen);
```

13.4 Forwarding Requests to a Queue

In some situations, a driver must requeue requests on its own, after it has received them from KMDF. For example, a driver might be able to respond to some device I/O control requests immediately, but might have to handle others at a later time. Because the requests are all of the same type (**WdfRequestTypeDeviceControl**), KMDF cannot deliver some to one queue and some to another. Instead, the driver must sort the requests as KMDF delivers them and place any that it cannot satisfy immediately into a manual, internal queue to handle later.

To forward a request to a queue, a KMDF driver calls **WdfRequest-ForwardToIoQueue**, passing as parameters the handle to the request and the handle to the queue. If the request is added successfully, KMDF returns the status **STATUS_SUCCESS**. A driver cannot return a request to the queue from which the driver most recently received it.

The **PCIDRV** sample uses this technique to delay processing **IOCTL_NDISPROT_INDICATE_STATUS** requests, as the following excerpt from its **EvtIoDeviceControl** callback (in **pcidrv.c**) shows:

```
case IOCTL_NDISPROT_INDICATE_STATUS:
  status = WdfRequestForwardToIoQueue (Request,
               fdoData->PendingIoctlQueue);
  ASSERT (status == STATUS_WDF_FORWARDED);
  break;
```

13.5 Retrieving Requests from a Manual Queue

When the driver is ready to handle a request from a manual queue, it calls a method on the queue object to retrieve one. A KMDF driver can

- Retrieve the next request from the queue.
- Retrieve the oldest request in the queue that pertains to a particular file object.
- Search the queue until it finds a particular request and then retrieve that request.

To remove the next item from a manual queue, a driver calls **WdfIo-QueueRetrieveNextRequest** with a handle to the queue and a pointer to a location to receive the handle to the request.

To remove the oldest request that specifies a particular file object, a driver calls **WdfIoQueueRetrieveRequestByFileObject**. The driver passes a handle to the file object along with the handle to the queue and a pointer to a location to receive the handle to the request. This method updates an internal queue pointer, so that the next time the driver calls it, it returns the next-oldest item, and so forth.

To search the queue for a particular request, the driver calls **WdfIoQueueFindRequest**. This method returns a handle to the request but does not remove the request from the queue. The driver can inspect the request to determine whether it is the one that the driver was seeking. If not, the request stays in the queue and the driver can search again. If so, the driver can dequeue the request by calling **WdfIoQueue-RetrieveFoundRequest**.

After the driver has removed a request from the queue, the driver "owns" that request. The driver must complete the request, forward it to another driver, or forward it to a different queue.

13.5.1 Code to Find a Request

The following function shows how the **PCIDRV** sample searches its manual device I/O control queue for a request with a particular function code and then retrieves that request. (The code is from the source file **pcidrv\sys\hw\nic_req.c**, and it has been slightly abridged.)

```
NICGETIoctlRequest (
       IN     WDFQUEUE       Queue,
       IN     ULONG          FunctionCode,
       OUT    WDFREQUEST*    Request
       )
   {
       NTSTATUS    status = STATUS_UNSUCCESSFUL
       WDF_REQUEST_PARAMETERS        params;
       WDFREQUEST                    tagRequest;
       WDFREQUEST                    prevTagRequest;

       WDF_REQUEST_PARAMETER_INIT (&params);

       *Request = NULL;
       prevTagRequest = tagRequest = NULL;

       do
       {
          WDF_REQUEST_PARAMETERS_INIT (&params);
          status = WdfIoQueueFindRequest (Queue,
                       prevTagRequest,
                       NULL,
                       &params,
                       &tagRequest);
```

```
            // WdfIoQueueFindRequest takes an extra reference on
            // the returned tagRequest to prevent the memory
            // being freed. However, the tagRequest is still
            // in the queue and can be canceled or removed by
            // another thread and completed.
            //

if (prevTagRequest)
{
      WdfObjectDereference (prevTagRequest);
}

if (status == STATUS_NO_MORE_ENTRIES)
{
      status = STATUS_UNSUCCESSFUL;
      break;
}

      if (status == STATUS_NOT_FOUND)
      {
         //
         // The prevTagRequest disappeared from the
         // queue for some reason - either it was
         // canceled or dispatched to the driver. There
         // might be other requests that match our
         // criteria so restart the search.
         //
         prevTagRequest = tagRequest = NULL;
         continue;
      }

      if (!NT_SUCCESS (status ))
      {
         status = STATUS_UNSUCCESSFUL;
         break;
      }

      if (FunctionCode ==
         params.Parameters.DeviceIoControl.IoControlCode)
      {
         status = WdfIoQueueRetrieveFoundRequest (
                     Queue,
                     tagRequest,      // TagRequest
                     Request
                  );
```

```
                WdfObjectDereference (tagRequest);

                if (status == STATUS_NOT_FOUND)
                {
                    //
                    // The TagRequest disappeared
                    // for some reason - either it was
                    // canceled or dispatched to the driver.
                    // Other requests might match our
                    // criteria so restart the search.
                    //
                    prevTagRequest = tagRequest = NULL;
                    continue;
                }

                if (!NT_SUCCESS (status))
                {
                    status = STATUS_UNSUCCESSFUL;
                    break;
                }

                //
                // Found a request. Drop the extra reference
                // before returning.
                //
                ASSERT (*Request == tagRequest);
                status = STATUS_SUCCESS;
                break;
            }
            else
            {
                //
                // This is not the request we need. Drop the
                // reference on the tagRequest after looking for
                // the next request.
                prevTagRequest = tagRequest;
                continue;
            }
        } WHILE (TRUE);
        return status;
    }
```

The sample driver starts by calling **WDF_REQUEST_PARAMETERS_INIT** to initialize a **WDF_REQUEST_PARAMETERS** structure. Later, when the driver calls **WdfIoQueueFindRequest**, KMDF returns the parameters for the request in this structure.

Next, the driver initializes the variables that it uses to keep track of the requests it has searched through. The variable **prevTagRequest** holds a handle to the previous request that the driver inspected, and **tagRequest** holds a handle to the current request. The driver initializes both values to **NULL** before it starts searching.

The search is conducted in a loop. Each time **NICGetIoctlRequest** calls **WdfIoQueueFindRequest**, it passes **prevTagRequest** to indicate where KMDF should start searching and passes a pointer to **tagRequest** to receive the handle to the current request. **WdfIoQueueFindRequest** also takes a handle to the queue, a handle to the related file object, and a pointer to the initialized **WDF_REQUEST_PARAMETERS** structure. The **PCIDRV** sample does not use file objects, so it passes **NULL** for the file object handle. Note that the driver reinitializes the **WDF_REQUEST_PARAMETERS** structure before each call, thus ensuring that it does not receive old data.

On the first iteration of the loop, **preTagRequest** is **NULL**. Therefore, the search starts at the beginning of the queue. **WdfIo-QueueFindRequest** searches the queue and returns the request's parameters (in the **Params** variable) and a handle to the request (in **tagRequest**). To prevent another component from deleting the request while the driver inspects it, **WdfIoQueueFindRequest** takes out a reference on request.

NICGetIoctlRequest compares the function code value that was returned in the request's parameters structure with the function code that the caller passed in. If the codes match, the driver calls **WdfIo-QueueRetrieveFoundRequest** to dequeue the request. **WdfIoQueue-RetrieveFoundRequest** takes three parameters: the handle to the queue, the handle returned by **WdfIoQueueFindRequest** that indicates which request to dequeue, and pointer to a location that will receive a handle to the dequeued request.

When **WdfIoQueueRetrieveFoundRequest** returns successfully, the driver "owns" the retrieved request. It deletes extra reference previously taken on the request by calling **WdfObjectDeference**. It then exits from the loop, and the **NICGetIoctlRequest** function returns a handle to the retrieved request. The caller can then perform the I/O operations that are required to satisfy the request.

If the function codes do not match, the driver sets **prevTagRequest** to **tagRequest** so that the search starts at the current location in the queue. However, the driver does not yet dereference the request object that **prevTagRequest** represents. It must maintain this reference until **WdfIoQueueFindRequest** has returned on the next iteration of the loop. The loop then executes again. This time, if **WdfIoQueueFindRequest** successfully finds a request, the driver deletes the reference that **WdfIoQueueFindRequest** acquired for the **prevTagRequest** and then compares the function codes as it did in the previous iteration.

If the request is no longer in the queue, **WdfIoQueueFindRequest** returns **STATUS_NOT_FOUND**. For example, the request might not be in the queue if it was canceled or was already retrieved by another thread. **WdfIoQueueRetrieveFoundRequest** can also return this same status if the handle passed in **tagRequest** is not valid. If either of these errors occurs, the driver restarts the search at the beginning. If either of these methods fails for any other reason, such as exhausting the queue, the driver exits from the loop.

13.6 Reading and Writing the Registry

KMDF includes numerous methods with which a driver can read and write the registry. These methods enable the driver to create, open, and close a registry key, and to query, change, and delete the values of keys and individual data items within them.

To read the value of a registry key, a driver opens the registry key and then calls a method that queries the registry for data. A driver can read the registry either before or after creating its device object.

To read the registry before creating the device object, a driver calls the **WdfFdoInitOpenRegistryKey** method with the following parameters:

- A handle to the **WDFDEVICE_INIT** structure that was passed to its **EvtDriverDeviceAdd** function.
- A **ULONG** value that identifies the key to open.
- A bit mask that indicates the type of required access.
- An optional attribute structure.
- A location to receive a handle to a **WDFKEY** object.

The **WDFDEVICE_INIT** structure contains settings that the framework requires before the device object has been created. **WdfFdoInit-OpenRegisterKey** requires some of this information so that it can find the requested key.

To read the registry after creating the device object, a driver calls **WdfDeviceOpenRegistryKey** with a handle to the device object instead of the **WDFDEVICE_INIT** structure.

To get the value of a single setting within the key, the driver must query the key. KMDF provides several query methods, each of which returns a different type of value. For example, if the information is stored as a **ULONG**, the driver calls **WdfRegisteryQueryUlong** with the handle to the keys, a pointer to the name of the value, and a pointer to a **ULONG** variable to receive the value. After completing its query, the driver closes the registry key by calling **WdfRegistryClose**.

13.6.1 Code to Read and Write the Registry

The **PCIDRV** sample provides functions that

- Read a **REG_DWORD** registry value that was written by another user mode or kernel mode component.
- Read a **REG_DWORD** registry value that was stored under the device key.
- Write a **REG_DWORD** registry value that was stored under the device key.

These functions are in the **pcidrv.c** source file.

The **PCIDRV** sample driver's **PciDrvReadFdoRegistryKeyValue** function is called from the **EvtDriverDeviceAdd** callback, before the driver creates the device object. It reads a key that the driver's **INF** wrote at installation, which indicates whether the driver was installed as an **NDIS** upper-edge miniport driver. This information is important because it determines whether the driver registers certain power policy and I/O event callbacks. If the driver was installed as an upper-edge miniport driver, it is not the power policy manager for its device; **NDIS** manages power policy.

The following is the source code for this function:

```
BOOLEAN
PciDrvReadFdoRegistryKeyValue (
   __in    PWDFDEVICE_INIT      DeviceInit,
   __in    PWCHAR               Name,
   __out  PULONG                Value
   )
{
   WDFKEY          hKey = NULL;
```

```
    NTSTATUS         status;
    BOOLEAN          retValue = FALSE;
    UNICODE_STRING   valueName;

    PAGED_CODE();

    *Value = 0;
    status = WdfFdoInitOpenRegistryKey (DeviceInit,
                 PLUGPLAY_REGKEY_DEVICE,
                 STANDARD_RIGHTS_ALL,
                 WDF_NO_OBJECT_ATTRIBUTES,
                 &hKey
                 );

    if (NT_SUCCESS (status))
    {
       RtlInitUnicodeString (&valueName, Name);
       status = WdfRegistryQueryULong (hKey,
                   &valueName,
                   Value
                   );
       if (NT_SUCCESS (status))
       {
          retValue = TRUE;
       }
       WdfRegistryClose (hKey);
    }
    return retValue;
}
```

First, the driver initializes the **Value** parameter that will receive the requested key value. Next, it opens the registry key by calling **WdfFdoInitOpenRegistryKey**. The **WDF_DEVICEINIT** object provides information that is required to identify the driver-specific key. The next parameter is a **ULONG** that contains flags that identify the key to open; the constant **PLUGPLAY_REGKEY_DEVICE** indicates the device's hardware key. Although the sample requests all access rights to the registry (**STANDARD_RIGHTS_ALL**), the driver only reads the key and does not write it, so **STANDARD_RIGHTS_READ** would also work. Finally, the driver specifies **WDF_NO_OBJECT_ATTRIBUTES** to indicate that it is not passing an attribute structure for the key object. The output parameter **hKey** receives a handle to the returned **WDFKEY** object.

If the driver successfully opens the hardware key, it can query the key for the requested value. The name of the value is passed into the current function as a pointer to a string. However, the KMDF query method requires the name in a counted Unicode string. Therefore, before querying for the value, the driver calls **RtInitUnicodeString** to copy the input string into a string of the correct format.

The driver then queries for the value by calling **WdfRegistryQueryUlong**, which returns the **ULONG** value of the key in the **Value** parameter. The driver then closes the key by calling **WdfRegisterClose** and the function returns.

The driver's other function to read the registry is similar. The only difference is that **PciDrvReadRegistryValue** is always called after the **WDFDEVICE** object has been created and therefore uses a handle to the **WDFDEVICE** object instead of a handle to the **WDFDEVICE_INIT** object.

The following is the code for this function:

```
BOOLEAN
PciDrvReadRegistryValue (
   __in    PFDO_DATA       FdoData,
   __in    PWCHAR          Name,
 __out     PULONG          Value
   )
{
   WDFKEY         hKey = NULL;
   NTSTATUS       status;
   BOOLEAN        retValue = FALSE;
   UNICODE_STRING valueName;

   PAGED_CODE ();

   *Value = 0;
   status = WdfDeviceOpenRegistryKey (FdoData->WdfDevice,
                PLUGPLAY_REGKEY_DEVICE,
                STANDARD_RIGHTS_ALL,
                WDF_NO_OBJECT_ATTRIBUTES,
                &hKey
                );
   if (NT_SUCCESS (status))
   {
      RtlInitUnicodeString (&valueName, Name);
      status = WdfRegistryQueryULong (hKey,
                  &valueName, Value
                  );
      if (NT_SUCCESS (status))
      {
         retValue = TRUE;
      }
```

```
        WdfRegistryClost (hKey);
    }
    return retValue;
}
```

This function is passed a pointer to the device context area (**FdoData**), where the driver keeps a handle to the device object. It passes the device object handle to the **WdfDeviceOpenRegistryKey** method. The driver then proceeds in exactly the same way as the previously discussed **PciDrvReadFdoRegisterKeyValue** function: It calls **RtlUnicode-StringCopyString** to build a Unicode string that holds the requested value name, calls **WdfRegistryQueryUlong** to get the **ULONG** value of the key, and calls **WdfRegistryClose** when it is finished.

The third registry function in the driver is **PciDrvWriteRegistry-Value**, which writes a **ULONG** value to the registry. This function differs from **PciDrvReadRegistryValue** in only two aspects:

- The value of the key is an input parameter to **PciDrvWrite-RegistryValue** but an output parameter to **PciDrvReadRegistry-Value**.
- **PciDrvWriteRegistryValue** calls **WdfRegistryAssignULong** to write a new value for the key, whereas **PciDrvReadRegistryValue** calls **WdfRegistryQueryULong** to read the current value of the key.

Otherwise, the functions are identical. The following statement shows how the driver writes the new value for the key:

```
status = WdfREgistryAssignULong (hKey, &valueName, Value);
```

Currently, the **PCIDRV** sample driver does not call **PciDrv-WriteRegistryValue**; this function is included only for demonstration.

13.7 WatchDog Timer: Self-Managed I/O

Some drivers perform I/O activities that are not related to queued I/O requests or must be synchronized with activities of WDM drivers in the same device stack. For example, a driver might maintain a timer that monitors the status of its device. Similarly, a driver might be required to communicate with its device or another driver at a particular point during the device's start-up or shut-down sequence. KMDF provides self-managed I/O to accommodate such requirements. The self-managed I/O callbacks correspond more closely to the underlying WDM Plug and Play and power management **IRPs** than do other WDF Plug and Play and power management callbacks.

To use self-managed I/O, a driver implements the self-managed I/O event callbacks. KMDF calls these callbacks during Plug and Play and power state transitions when the device is added to or removed from the system, when the device is stopped to rebalance resources, when the idle device transitions to a low-power state, and when the device returns to the working state from a low-power idle state.

The following are the self-managed I/O callbacks:

- **EvtDeviceSelfManagedIoInit**
- **EvtDeviceSelfManagedIoSuspend**
- **EvtDeviceSelfManagedIoFlush**
- **EvtDeviceSelfManagedIoCleanup**
- **EvtDEviceSelfManagedIoRestart**

13.7.1 Self-Managed I/O Device Startup and Restart

When the system is booted or the user plugs in the device, KMDF calls the driver's **EvtDeviceSelfManagedIoInit** callback after the driver's **EvtDeviceDOEntry** function has returned but before KMDF completes the underlying Plug and Play or power **IRP**. KMDF calls this function only during the initial start-up sequence; it does not call this function when the device returns to the working state from a low-power state.

The **EvtDeviceSelfManagedIoInit** callback should perform whatever tasks are required to initiate the I/O that the framework doesn't manage. For example, a driver that must monitor the state of its device might initialize and start a timer.

When the device returns to the working state from a low-power state, such as occurs when the device has been idle or has been stopped to rebalance resources, KMDF calls the **EvtDeviceSelfManagedIoRestart** callback.

Like the self-managed I/O initialization callback, this function is the last one that is called after the device returns to the working state, but before

WDF completes the underlying **IRP**. **EvtDeviceSelfManaged-IoRestart** should resume any I/O activities that **EvtDeviceSelfManaged-IoInit** initialized and that were later suspended when the device exited from the working state. Typically, this means that it reverses the actions of **EvtDeviceSelfManagedIoSuspend.**

EvtDeviceSelfManagedIoRestart is called only after **EvtDevice-SelfManagedIoSuspend** has previously been called. This function is called only when the device has been in a low-power state or its resources have been rebalanced; it is not called when the user initially plugs in the device.

13.7.2 Self-Managed I/O During Device Power-Down and Removal

When the device is powered down or removed, KMDF calls one or more of the self-managed I/O callbacks so that the driver can stop and clean up after its self-managed I/O operations.

Every time the device goes through the power-down sequence—whether because it is idle, it is being removed, or system resources are being rebalanced—KMDF calls the **EvtDeviceSelfManagedIo-Suspend** callback. This function should stop any self-managed I/O activities that are in progress and must be handled while the device is present. During rebalance, power-down, and orderly removal, it is called while the device is still operational, before **EvtDeviceDOExit**. During surprise removal, it is called before **EvtDeviceSurpriseRemoval** if the device was in a low-power state and afterward if the device was in the **DO** state.

If the device is being removed, KMDF calls **EvtDevice-SelfManagedIoFlush** after the device has been stopped. This function should fail any I/O requests that the driver did not complete before the device was removed. It is called after the driver's **EvtDEvice-SelfManagedIoSuspend** and **EvtDeviceDOExit** functions have returned.

Finally, KMDF calls **EvtDeviceSelfManagedIoCleanup** after device removal is complete. This function should ensure that all self-managed I/O has stopped completely and should release any resources that **EvtDEviceSelfManagedIoInit** allocated for self-managed I/O. The clean-up function is called only once.

13.7.3 Implementing a Watchdog Timer

The **PCIDRV** sample uses self-managed I/O to implement a watchdog timer, which is used during hardware link detection to check for hangs. Implementing the watchdog timer involves the following driver tasks:

- Setting callbacks for the self-managed I/O events.
- Initializing the timer in **EvtDeviceSelfManagedIoInit**.
- Stopping the timer in **EvtDeviceSelf-ManagedIoSuspend**.
- Restarting the timer in **EvtDeviceSelfManagedIoRestart**.
- Deleting the timer and resources in **EvtDeviceSelfManaged-IoCleanup**.

The **PCIDRV** sample does not implement the **EvtDeviceSelf-ManagedIoFlush** callback because no I/O requests are involved in its self-managed I/O. The suspend and clean-up callbacks are sufficient.

A driver registers its self-managed I/O callbacks by setting their entry points in the **WDF_PNP_POWER_CALLBACKS** structure along with the other Plug and Play and power event callbacks (such as **EvtDEviceDOEntry** and **EvtDeviceDOExit**, among others). The driver sets these in the **EvtDriverDeviceAdd** callback, before it creates the **WDFDEVICE** object.

13.7.3.1 Code to Set Self-Managed I/O Callbacks

The **PCIDRV** sample registers these callbacks in the **PciDrvEvtDevice-Add** function in **pcidrv.c**:

```
WDF_PNPPOWER_EVENT_CALLBACKS  pnpPowerCallbacks;

//
// Initialize the PnpPowerCallbacks structure.
//
WDF_PNPPOWER_EVENT_CALLBACK_INIT (&pnpPowerCallbacks);

//
// Set entry points for self-managed I/O callbacks.
//
pnpPowerCallbacks.EvtDeviceSElfManagedIoInit       =
   PciDrvEvtDEvicesSelfManagedIoInit;
pnpPowerCallbacks.EvtDeviceSelfManagedIoCleanup    =
   PciDrvEvtDeviceSelfManagedIoCleanup;
```

```
pnpPowerCallbacks.EvtDeviceSelfManagedIoSuspend    =
   PciDrvEvtDeviceSelfManagedIoSuspend;
pnpPowerCallbacks.EvtDeviceSelfManageIoRestart      =
   PciDrvEvtDeviceSelfManagedIoRestart;
//
// Register the PnP and power callbacks.
//
WdfDeviceInitSetPnpPowerEventCallbacks (DeviceInit,
   &pnpPowerCallbacks);
```

As the example shows, the **PCIDRV** sample sets callbacks for **EvtDeviceSelfManagedIoInit**, **EvtDeviceSelfManagedIoCleanup**, **EvtDeviceSelfManagedIoSuspend**, and **EvtDeviceSelfManagedIo-Restart**.

13.7.3.2 Code to Create and Initialize the Timer

The **PCIDRV** sample creates, initializes, and starts the watchdog timer in its **EvtDeviceSelfManagedIoInit** callback. The watchdog timer is a WDF time object (**WDFTIMER**). When the timer expires, KMDF queues a **DPC**, which calls the driver's **EvtTimerFunc.**

The following code is the sample's **EvtDeviceSelfManagedIoInit** callback, which appears in the **pcidrv.c** source file:

```
NTSTATUS
PciDrvEvtDeviceSelfManagedIoInit (
   IN   WDFDEVICE   Device
   )
{
   PFDO_DATA           fdoData = NULL;
   WDF_TIMER_CONFIG    wdfTimerconfig;
   NTSTATUS            status;
   WDF_OBJECT_ATTRIBUTES   timerAttributes;

   PAGED_CODE ();

   TraceEvents (TRACE_LEVEL_INFORMATION, DBG_PNP,
         "--> PciDrvEvtDeviceSelfManagedIoInit\n");

   fdoData = FdoGetData (Device);

//
// To minimize init-time, create a timer DPC to do link
// detection. The DPC will also be used to check for hardware
// hang.
WDF_TIMER_CONFIG_INIT (&wdfTimerconfig,
      NICWatchDogEvtTimerFunc);

WDF_OBJECT_ATTRIBUTES_INIT (&timerAttributes);
timerAttributes.ParentObject = fdoData->WdfDevice;

status = WdfTimerCreate (
            &wdfTimerConfig,
            &timerAttributes,
            &fdoData->WatchDogTimer
            );

if (!NT_SUCCESS (status))
{
   TraceEvents (TRACE_LEVEL_ERROR, DBG_PNP,
      "Error: WdfTimerCreate create failed 0x%x\n",
      Status);
   return status;
}

NICStartWatchDogTimer (fdoData);

TraceEvents (TRACE_LEVEL_INFORMATION, DBG_PNP,
      "<-PciDrvEvtDeviceSelfManagedIoInit\n");

return status;
}
```

The driver declares two structures for use in creating the timer: a **WDF_TIMER_CONFIG** structure named **wdfTimerConfig** and a **WDF_OBJECT_ATTRIBUTES** structure named **timerAttributes**. To initialize the **WdfTimerConfig** structure, the driver uses the **WDF_TIMER_CONFIG_INIT** function, passing as parameters the **WdfTimerConfig** structure and a pointer to **NICWatchdogEvtTimer-Func**, which is the driver's **EvtTimerFunc** callback.

Next, the driver initializes the attribute's structure by using the **WDF_OBJECT_ATTRIBUTES_INIT** function. By default, a timer object has no parent, so the driver sets the **ParentObject** field to the device object (**WdfDevice**) so that KMDF deletes the timer when it deletes the device object.

Finally, the driver calls **WdfTimerCreate** to create the timer, passing as parameters the configuration structure, the attribute's structure, and a location to receive the handle to the timer. If KMDF successfully creates the timer, **PCIDRV** calls the internal function **NICStartWatchDog-Timer** to start the timer.

13.7.3.3 Code to Start the Timer

The **NICStartWatchDogTimer** function appears in the **sys/HW/isrdpc.c** source file, as follows:

```
VOID
NICStartWatchDogTimer (
    IN    PFDO_DATA       FdoData
    )
{
    LARGE_INTEGER         dueTime;

    if (!FdoData->CheckForHang)
    {
        //
        // Set the link detection flag to indicate that
        // NICWatchDogEvtTimerFunc
        // is first doing link detection.
        //
        MP_SET_FLAG (FdoData,
            fMP_ADAPTER_LINK_DETECTION);
        FdoData->CheckforHang = FALSE;
        FdoData->bLinkDetectionWait = FALSE;
        FdoData->bLookForLink = FALSE;
        dueTime.QuadPart = NIC_LINK_DETECTION_DELAY;
    }
    else
    {
        dueTime.QuadPart = NIC_CHECK_FOR_HANG_DELAY;
    }
    WdfTimerStart (FdoData->WatchDogTimer,
                      dueTime.QuadPart
                      );
        return;
    }
```

This function sets the expiration time for the timer to a hardware-dependent value, depending on whether the driver is attempting to detect a link or check for a device hang. It then starts the timer by calling **WdfTimerStart**. When the timer expires, KMDF queues a DPC that invokes the driver's timer function, **NICWatchDogEvtTimerFunc**. The timer function performs the required task (link detection or check for hang) and then restarts the timer by calling **WdfTimerStart** in the same way shown in the preceding example.

13.7.3.4 Code to Stop the Timer

When the device leaves the working state or is removed from the system, KMDF calls the **EvtDeviceSelfManagedIoSuspend** callback. In the **PCIDRV** sample, this callback stops the timer, as the following code from **pcidrv.c** shows:

```
NTSTATUS
PciDrvEvtDeviceSelfManagedIoSuspend (
   IN    WDFDEVICE    Device
   )
{
   PFDO_DATA    fdoData = NULL;

   PAGED_CODE ();

   fdoData = FdoGetData (Device);

   //
   // Stop the watchdog timer and wait for DPC to run
   // to completion
   // if it has already fired.
   //
   WdfTimerStop (fdoData->WatchDogTimer, TRUE);
   return STATUS_SUCCESS;
}
```

To stop the timer, the driver simply calls **WdfTimerStop**, passing as parameters the handle to the timer and a Boolean value. The **PCIDRV** sample passes **TRUE** to specify that if the driver has any **DPC**s in the **DPC** queue (including the **NICWatchDogEvtTimerFunc** timer **DPC** function), KMDF should wait until all of those functions have returned before stopping the timer. Specifying **FALSE** means that KMDF should stop the timer immediately without waiting for any **DPC**s to complete.

WdfTimerStop is defined as a Boolean function, which returns **TRUE** if the timer object was in the system's timer queue. However, the **PCIDRV** sample does not check the return value because it waits for all the driver's **DPC**s to complete, so whether the timer was already set is not important.

13.7.3.5 Code to Restart the Timer

When the device returns to the working state after being in low-power state, KMDF calls the **EvtDeviceSelfManagedIoRestart** callback.

In the **PCIDRV** driver, this callback restarts the timer as follows, from the **pcidrv.c** source file:

```
NTSTATUS
PciDrvEvtDeviceSelfManagedIoRestart (
    IN    WDFDEVICE    Device
    )
{
    PFDO_DATA    fdoData;

    PAGED_CODE ();

    fdoData = FdoGetData (Device);

    //
    // Restart the watchdog timer.
    //
    NICStartWatchDogTimer (fdoData);
    return STATUS_SUCCESS;
}
```

Restarting the timer simply requires a call to the internal **NICStartWatchDogTimer**, as previously discussed. Because the device object and the timer (a child of the device object) were not deleted when the device transitioned out of the working state, the driver is not required to reinitialize or recreate the timer object.

13.7.5.6 Code to Delete the Timer

When the device is removed, the driver deletes the timer in its **EvtDeviceSelfManagedIoCleanup** function, as follows:

```
VOID
PciDrvEvtDeviceSelfManagedIoCleanup (
    IN    WDFDEVICE    Device
    )
{
    PFDO_DATA fdoData = NULL;

    PAGED_CODE ();

    fdoData = FdoGetData (Device);

    if (fdoData->WatchDogTimer)
    {
        WdfObjectDelete (fdoData->WatchDogTimer);
    }
    return;
}
```

To delete the timer, the driver simply calls **WdfObjectDelete**, passing a handle to the timer object. If the driver had allocated any additional resources related to the timer, it would release those resources in this function.

This page intentionally left blank

APPENDIX

DRIVER INFORMATION WEB SITES

This appendix contains a list of relevant topics available on the Internet. These were accurate and active at the time this book was written; however, URLs change and some may no longer be active.

1. Using WinUSB for User-Mode to USB Device Communication
   http://www.osronline.com/article.cfm?article=532
2. Getting Started with Windows Drivers
   http://go.microsoft.com/fwlink/?LinkId=79284
3. Kernel-Mode Driver Architecture
   http://go.microsoft.com/fwlink/?LinkId=79288
4. Device and Driver Installation
   http://go.microsoft.com/fwlink/?LinkId=79294
5. Driver Developer Tools
   http://go.microsoft.com/fwlink/?LinkId=79298
6. Using Checked Builds of Windows
   http://go.microsoft.com/fwlink/?LinkId=79304
7. Download Windows Symbol Packages
   http://go.microsoft.com/fwlink/?LinkId=79331
8. Windows Driver Foundation (WDF)
   http://go.microsoft.com/fwlink/?LinkId=79335
9. About WDK and Developer Tools
   http://go.microsoft.com/fwlink/?LinkId=79337
10. Getting Started with Driver Development
    http://go.microsoft.com/fwlink/?LinkId=79338
11. Managing Hardware Priorities
    http://go.microsoft.com/fwlink/?LinkId=79339
12. User-Mode Driver Framework Design Guide
    http://go.microsoft.com/fwlink/?LinkId=79341

13. Kernel-Mode Driver Framework Design Guide
    http://go.microsoft.com/fwlink/?LinkId=79342
14. Installing UMDF Drivers
    http://go.microsoft.com/fwlink/?LinkId=79345
15. Building and Loading a Framework-Based Driver
    http://go.microsoft.com/fwlink/?LinkId=79347
16. Building Drivers
    http://go.microsoft.com/fwlink/?LinkId=79348
17. Build Utility Reference
    http://go.microsoft.com/fwlink/?LinkId=79349
18. Utilizing a Sources File Template
    http://go.microsoft.com/fwlink/?LinkId=79350
19. Roadmaps
    http://go.microsoft.com/fwlink/?LinkId=79351
20. Installing a Framework-Based Driver
    http://go.microsoft.com/fwlink/?LinkId=79352
21. Installation and Driver Signing—Papers
    http://go.microsoft.com/fwlink/?LinkId=79354
22. Framework Library Versions
    http://go.microsoft.com/fwlink/?LinkId=79355
23. Driver Signing Requirements for Windows
    http://go.microsoft.com/fwlink/?LinkId=79358
24. Windows Logo Program—Overview
    http://go.microsoft.com/fwlink/?LinkId=79359
25. Creating a Catalog file for a PnP Driver Package
    http://go.microsoft.com/fwlink/?LinkId=79360
26. Code-Signing Best Practices
    http://go.microsoft.com/fwlink/?LinkId=79361
27. Distributing Drivers on Windows Update
    http://go.microsoft.com/fwlink/?LinkId=79362
28. Kernel-Mode Code Signing Walkthrough
    http://go.microsoft.com/fwlink/?LinkId=79363
29. How Setup Selects Drivers—PnP Manager
    http://go.microsoft.com/fwlink/?LinkId=79364
30. Driver Installation Rules
    http://go.microsoft.com/fwlink/?LinkId=79365
31. Using Driver Install Frameworks (DIFx)
    http://go.microsoft.com/fwlink/?LinkId=79366

32. Writing a Device Installation Application
    http://go.microsoft.com/fwlink/?LinkId=79367
33. Troubleshooting Device and Driver Installation
    http://go.microsoft.com/fwlink/?LinkId=79370
34. Guidelines for Using SetAPI
    http://go.microsoft.com/fwlink/?LinkId=79371
35. Hardware and Driver Developer Blogs
    http://go.microsoft.com/fwlink/?LinkId=79579
36. Hardware and Driver Developer Community
    http://go.microsoft.com/fwlink/?LinkId=79580
37. UMDF Objects and Interfaces
    http://go.microsoft.com/fwlink/?LinkId=79583
38. Kernel-Mode Driver Framework Objects
    http://go.microsoft.com/fwlink/?LinkId=79584
39. Driver Verifier
    http://go.microsoft.com/fwlink/?LinkId=79588
40. Microsoft Application Verifier
    http://go.microsoft.com/fwlink/?LinkId=79601
41. How do I keep my driver from running out of kernel-mode stack?
    http://go.microsoft.com/fwlink/?LinkId=79604
42. Component Object Model
    http://go.microsoft.com/fwlink/?LinkId=79770
43. Inside COM
    http://go.microsoft.com/fwlink/?LinkId=79771
44. ATL (Active Template Library)
    http://go.microsoft.com/fwlink/?LinkId=79772
45. Installing Just the Checked Operating System and HAL
    http://go.microsoft.com/fwlink/?LinkId=79774
46. ChkIN—INF Syntax Checker
    http://go.microsoft.com/fwlink/?LinkId=79776
47. DevCon WDK Command-Line Tool
    http://go.microsoft.com/fwlink/?LinkId=79777
48. Device Path Exerciser
    http://go.microsoft.com/fwlink/?LinkId=79778
49. KrView—The Kernrate Viewer
    http://go.microsoft.com/fwlink/?LinkId=79779
50. Plug and Play Driver Test
    http://go.microsoft.com/fwlink/?LinkId=79780

51. PNPCPU
    http://go.microsoft.com/fwlink/?LinkId=79781
52. PoolMon
    http://go.microsoft.com/fwlink/?LinkId=79782
53. PwrTest
    http://go.microsoft.com/fwlink/?LinkId=79783
54. Windows Device Testing Framework
    http://go.microsoft.com/fwlink/?LinkId=79785
55. Verifier Command Line
    http://go.microsoft.com/fwlink/?LinkId=79788
56. Driver Verifier Manager
    http://go.microsoft.com/fwlink/?LinkId=79789
57. Debugging a Framework-Based Driver
    http://go.microsoft.com/fwlink/?LinkId=79790
58. Windows Error Reporting: Getting Started
    http://go.microsoft.com/fwlink/?LinkId=79792
59. Handing Driver Failures—UMDF drivers
    http://go.microsoft.com/fwlink/?LinkId=79794
60. Scanning the Driver
    http://go.microsoft.com/fwlink/?LinkId=80057
61. Adding the Reflector
    http://go.microsoft.com/fwlink/?LinkId=80058
62. Build Utility Limitations and Rules
    http://go.microsoft.com/fwlink/?LinkId=80059
63. C++ for Kernel Mode Drivers: Pros and Cons
    http://go.microsoft.com/fwlink/?LinkId=80060
64. Can I customize DoTraceMessage?
    http://go.microsoft.com/fwlink/?LinkId=80061
65. Creating Reliable and Secure Drivers
    http://go.microsoft.com/fwlink/?LinkId=80063
66. Debugging Tools for Windows—Overview
    http://go.microsoft.com/fwlink/?LinkId=80065
67. Defect Viewer
    http://go.microsoft.com/fwlink/?LinkId=80066
68. Device Manager Error Messages
    http://go.microsoft.com/fwlink/?LinkId=80068
69. DllMain Callback Function
    http://go.microsoft.com/fwlink/?LinkId=80069

70. DMA Verification
    http://go.microsoft.com/fwlink/?LinkId=80070
71. Waits and APCs
    http://go.microsoft.com/fwlink/?LinkId=80071
72. Handling DMA Operations in Framework-Based Drivers
    http://go.microsoft.com/fwlink/?LinkId=80073
73. Interface Definition Language (IDL) File
    http://go.microsoft.com/fwlink/?LinkId=80074
74. Library Processing in Static Driver Verifier
    http://go.microsoft.com/fwlink/?LinkId=80077
75. PREfast for Drivers
    http://go.microsoft.com/fwlink/?LinkId=80079
76. Results Pane in the Static Driver Verifier Report
    http://go.microsoft.com/fwlink/?LinkId=80081
77. Static Driver Verifier—WHDC Web Site
    http://go.microsoft.com/fwlink/?LinkId=80082
78. Static Driver Verifier—WDK Documentation
    http://go.microsoft.com/fwlink/?LinkId=80084
79. Static Driver Verifier Commands
    http://go.microsoft.com/fwlink/?LinkId=80085
80. Static Driver Verifier Limitations
    http://go.microsoft.com/fwlink/?LinkId=80086
81. Uninstalling Drivers and Devices
    http://go.microsoft.com/fwlink/?LinkId=80089
82. WPP Software Tracing
    http://go.microsoft.com/fwlink/?LinkId=80090
83. Preparing to Run Static Driver Verifier
    http://go.microsoft.com/fwlink/?LinkId=80606
84. Build (Build Utility)
    http://go.microsoft.com/fwlink/?LinkId=80609
85. Thorough Static Analysis of Device Drivers
    http://go.microsoft.com/fwlink/?LinkId=80612
86. Handling I/O Requests in Framework-Based Drivers
    http://go.microsoft.com/fwlink/?LinkId=80613
87. Object Names
    http://go.microsoft.com/fwlink/?LinkId=80615
88. ACCESS_MASK
    http://go.microsoft.com/fwlink/?LinkId=80616

89. Boot Options for Driver Testing and Debugging
    http://go.microsoft.com/fwlink/?LinkId=80622
90. Trace Message Prefix
    http://go.microsoft.com/fwlink/?LinkId=80623
91. Securing Device Objects
    http://go.microsoft.com/fwlink/?LinkId=80624
92. Creating Secure Device Installations
    http://go.microsoft.com/fwlink/?LinkId=80625
93. SDDL for Device Objects
    http://go.microsoft.com/fwlink/?LinkId=80626
94. Synchronization
    http://go.microsoft.com/fwlink/?LinkId=80899
95. How to: Specify Additional Code Information
    http://go.microsoft.com/fwlink/?LinkId=80906
96. Using a Pragma Warning Directive
    http://go.microsoft.com/fwlink/?LinkId=80908
97. Developing Drivers with WDF
    http://go.microsoft.com/fwlink/?LinkId=80911
98. Boot Configuration Data
    http://go.microsoft.com/fwlink/?LinkId=80914
99. Device Interface Classes
    http://go.microsoft.com/fwlink/?LinkId=81577
100. Using Device Interfaces
    http://go.microsoft.com/fwlink/?LinkId=81578
101. Controlling Device Access in Framework-Based Drivers
    http://go.microsoft.com/fwlink/?LinkId=81579
102. Implementing WMI
    http://go.microsoft.com/fwlink/?LinkId=81581
103. Specifying Priority Boosts When Completing I/O Requests
    http://go.microsoft.com/fwlink/?LinkId=81582
104. Writing a Bug Check Callback Routine
    http://go.microsoft.com/fwlink/?LinkId=81587
105. KeRegisterNmiCallback
    http://go.microsoft.com/fwlink/?LinkId=81588
106. HAL Library Routines
    http://go.microsoft.com/fwlink/?LinkId=81591
107. Using Device Installation Functions
    http://go.microsoft.com/fwlink/?LinkId=82107

108. Using GUIDs in Drivers
http://go.microsoft.com/fwlink/?LinkId=82109
109. PnP and Power Management in Framework-Based Drivers
http://go.microsoft.com/fwlink/?LinkId=82110
110. USB Power Management
http://go.microsoft.com/fwlink/?LinkId=82114
111. Plug and Play—Architecture and Driver Support
http://go.microsoft.com/fwlink/?LinkId=82116
112. Managing Kernel Objects
http://go.microsoft.com/fwlink/?LinkId=82272
113. Device Management
http://go.microsoft.com/fwlink/?LinkId=82273
114. Introduction to UMDF
http://go.microsoft.com/fwlink/?LinkId=82316
115. Getting Started with Kernel-Mode Driver Framework
http://go.microsoft.com/fwlink/?LinkId=82317
116. Service User Accounts
http://go.microsoft.com/fwlink/?LinkId=82318
117. WDM to KMDF Porting Guide
http://go.microsoft.com/fwlink/?LinkId=82319
118. Interpreting Bug Check Codes
http://go.microsoft.com/fwlink/?LinkId=82820
119. I/O Completion/Cancellation Guidelines
http://go.microsoft.com/fwlink/?LinkId=82321
120. Introduction to WMI
http://go.microsoft.com/fwlink/?LinkId=82322
121. ExAllocatePoolWithTag
http://go.microsoft.com/fwlink/?LinkId=82323
122. Locks, Deadlocks, and Synchronization
http://go.microsoft.com/fwlink/?LinkId=82717
123. Operating Systems, Stallings
http://go.microsoft.com/fwlink/?LinkId=82718
124. Locking Pageable Code or Data
http://go.microsoft.com/fwlink/?LinkId=82719
125. Microsoft Windows Internals Fourth Edition
http://go.microsoft.com/fwlink/?LinkId=82721
126. PAGED_CODE
http://go.microsoft.com/fwlink/?LinkId=82722

127. PAGED_CODE_LOCKED
http://go.microsoft.com/fwlink/?LinkId=82723
128. SECURITY_IMPERSONATE_LEVEL Enumeration
http://go.microsoft.com/fwlink/?LinkId=82952
129. Specifying WDF Directives in INF Files
http://go.microsoft.com/fwlink/?LinkId=82953
130. WINUSB_SETUP_PACKET
http://go.microsoft.com/fwlink/?LinkId=83355
131. Event Tracing
http://go.microsoft.com/fwlink/?LinkId=84477
132. Engineering Windows 7 Blog
http://blogs.msdn.com/e7
133. Microsoft Developer Network
http://msdn.microsoft.com
134. Microsoft Hardware Newsletter
http://www.microsoft.com/whdc
135. Microsoft Research: SLAM—Automatically Checks C-Based Programs
http://research.microsoft.com/slam
136. Obtain the WDK version 7.0.0
http://www.microsoft.com/whdc
137. Using WinUSB for User-Mode to USB Device Communication
http://www.osronline.com/article.cfm?article=532
138. The Windows Blog
http://windowsteamblog.com/blogs/developers/default.aspx
139. The Windows 7 Team Blog
http://windowsteamblog.com/blogs/windows7/default.aspx
140. Windows 7 Device Drivers Available for Download
www.blogsdna.com/2462/official_windows_7_device_drivers_available_for_download.htm
141. Windows 7 Sensor Windows Driver Kit
www.pctipsbox.com/windows_7_sensor_windows.driver_kit
142. Windows 7 WDK—New for Device and Driver Installation
msdn.microsoft.com/en-us/library/dd835060.aspx

Bibliography

Anderson, Don. *Universal Serial Bus System Architecture, Second Edition*. Addison-Wesley, 2001. ISBN 0-201-30975-0

Booch, Grady. *Object-Oriented Analysis and Design with Applications, Third Edition*. Addison-Wesley, 2007. ISBN 0-201-89551-X

Deitel, Paul J., and Harvey M. Deitel. *C++ How to Program, Seventh Edition*. Prentice Hall, 2009. ISBN 0-13-611726-0

Deitel, Paul J., and Harvey M. Deitel. *C How to Program, Sixth Edition*. Prentice Hall, 2009. ISBN 0-13-612356-2

Gamma, Erich, and Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2

Hewardt, Mario, and Daniel Pravat. *Advanced Windows Debugging*. Addison-Wesley, 2007. ISBN 0-321-37446-0

Oney, Walter. *Programming the Microsoft Windows Driver Model, Second Edition*. Microsoft Press, 2002. ISBN 0-73-561803-8

Rogerson, Dale. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997. ISBN 1-57-231349-8

Russinovich, Mark E., and David A. Solomon. *Windows Internals, Fifth Edition*. Microsoft Press, 2009. ISBN 0-73-562530-1

Shanley, Tom, and Don Anderson. *PCI System Architecture, Fourth Edition*. Addison-Wesley, 1999. ISBN 0-201-30974-2

This page intentionally left blank

INDEX