Microsoft

DEVELOPING DRIVERS WITH THE WINDOWS® DRIVER FOUNDATION

Penny Orwick
Guy Smith

Microsoft

Developing Drivers with the Windows® Driver Foundation

Penny Orwick
Guy Smith

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2007922586

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 2 1 0 9 8 7

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Developmental Editor: Devon Musgrave
Project Editor: Denise Bankaitis

Body Part No. X13-62472

Contents at a Glance

Part 1 **Getting Started with WDF**

1 Introduction to WDF . . . 3
2 Windows Driver Fundamentals . . . 23
3 WDF Fundamentals . . . 51

Part 2 **Exploring the Frameworks**

4 Overview of the Driver Frameworks . . . 67
5 WDF Object Model . . . 91
6 Driver Structure and Initialization . . . 129

Part 3 **Applying WDF Fundamentals**

7 Plug and Play and Power Management . . . 165
8 I/O Flow and Dispatching . . . 223
9 I/O Targets . . . 307
10 Synchronization . . . 379
11 Driver Tracing and Diagnosability . . . 411
12 WDF Support Objects . . . 441
13 UMDF Driver Template . . . 475

Part 4 **Additional Topics for KMDF Drivers**

14 Beyond the Frameworks . . . 497
15 Scheduling, Thread Context, and IRQL . . . 507
16 Hardware Resources and Interrupts . . . 529
17 Direct Memory Access . . . 549
18 An Introduction to COM . . . 583

Part 5 Building, Installing, and Testing a WDF Driver

19 How to Build WDF Drivers. 617
20 How to Install WDF Drivers 635
21 Tools for Testing WDF Drivers. 667
22 How to Debug WDF Drivers 697
23 PRE*f*ast for Drivers 731
24 Static Driver Verifier 823

Table of Contents

Foreword . . . xxiii
Acknowledgments . . . xxvii

Part 1 **Getting Started with WDF**

1 Introduction to WDF . . . 3

About This Book . . . 4
Who Should Read This Book . . . 4
About Part 1: Getting Started with WDF . . . 5
About Part 2: Exploring the Frameworks . . . 6
About Part 3: Applying WDF Fundamentals . . . 6
About Part 4: Digging Deeper: More Topics for WDF Drivers . . . 8
About Part 5: Building, Installing, and Testing a WDF Driver . . . 9
Conventions Used in This Book . . . 10
Getting Started with Driver Development . . . 11
System Requirements for Driver Development . . . 12
How to Obtain and Install the WDK . . . 13
WDK Libraries . . . 14
WDK Documentation . . . 14
WDK Tools . . . 15
WDK Samples . . . 16
How to Obtain Checked Builds of Windows . . . 17
How to Obtain Debugging Tools . . . 18
How to Obtain OSR Learning Devices . . . 19
Key Information Sources . . . 20
Key References . . . 22

2 Windows Driver Fundamentals 23

What Is a Driver? 24

Core Windows Architecture 25

Driver Architecture 27

Kernel Objects and Data Structures 31

The Windows I/O Model 31

I/O Requests 32

How a Device Stack Handles IRPs 33

Data Buffers and I/O Transfer Types 34

How to Transfer Data to or from a Device 35

About Plug and Play and Power Management 36

Basic Kernel-Mode Programming 36

Interrupts and IRQLs 37

Concurrency and Synchronization 39

Memory 42

Tips for Programming in Kernel Mode 44

A Basic Vocabulary 47

3 WDF Fundamentals 51

WDF and WDM 51

What Is WDF? 52

WDF Object Model 53

Programming Interface 54

Object Hierarchy 54

Concurrency and Synchronization 55

I/O Model 55

I/O Request Cancellation 56

I/O Targets 57

How to Handle Nonfatal Errors 58

Plug and Play and Power Management 59

Security 61

Safe Defaults 62

Parameter Validation 62

WDF Verification, Tracing, and Debugging Support 63

Serviceability and Versioning 63

Part 2 Exploring the Frameworks

4 Overview of the Driver Frameworks 67

The Frameworks: An Overview 67
UMDF Overview 68
UMDF Framework Objects 69
UMDF Callback Objects 70
KMDF Overview 71
KMDF Objects 71
KMDF Callback Functions 72
WDF Architecture 73
UMDF Infrastructure 76
UMDF Infrastructure Components 77
Fatal Errors in UMDF Drivers 80
A Typical UMDF I/O Request 80
KMDF Infrastructure 81
KMDF Infrastructure Components 82
Fatal Errors in KMDF Drivers 83
A Typical KMDF I/O Request 84
Device and Driver Support in WDF 84
Devices Supported by UMDF 85
Devices Supported by KMDF 88
Choosing the Right Framework 89

5 WDF Object Model 91

Overview of the Object Model 92
About Methods, Properties, and Events 92
About Event Callbacks 93
About Object Attributes 94
About Object Hierarchy and Lifetime 94
About Object Context 95
UMDF Object Model Implementation 95
UMDF Naming Conventions 95
UMDF Framework Objects and Interfaces 97
UMDF Driver Callback Objects and Interfaces 99
UMDF Example: Objects and Callback Interfaces 101

KMDF Object Model Implementation . . . 102
KMDF Object Types . . . 102
KMDF Naming Conventions . . . 104
Object Creation . . . 106
UMDF Object Creation . . . 106
KMDF Object Creation . . . 107
Object Hierarchy and Lifetime . . . 110
UMDF Object Hierarchy . . . 112
KMDF Object Hierarchy . . . 114
Object Deletion . . . 115
Object Context Areas . . . 122
UMDF Object Context Data . . . 122
KMDF Object Context Area . . . 126

6 Driver Structure and Initialization . . . 129

Required Driver Components . . . 129
UMDF Driver Structure and Requirements . . . 130
KMDF Driver Structure and Requirements . . . 133
Driver Object . . . 135
UMDF Driver Callback Object Creation . . . 135
KMDF Driver Object Creation . . . 137
Device Objects . . . 140
Types of Device Objects . . . 140
WDF Drivers, Driver Types, and Device Object Types . . . 144
Device Properties . . . 145
Device Object Initialization . . . 146
Queues and Other Support Objects . . . 147
Device Interfaces . . . 148
UMDF Device Object Creation and Initialization . . . 149
Device Callback Object Creation . . . 149
Framework Device Object Creation and Initialization . . . 150
UMDF Example: Device Interface . . . 152
KMDF Device Object Creation and Initialization . . . 152
KMDF Device Initialization Structure . . . 153
Device Object Context Area . . . 154
KMDF Device Object Creation . . . 155
Additional EvtDriverDeviceAdd Tasks . . . 155
KMDF Example: EvtDriverDeviceAdd Callback Function . . . 155

Child Device Enumeration (KMDF PDOs Only). 157
Static and Dynamic Enumeration in Bus Drivers. 157
PDO-Specific Initialization 158
Device Naming Techniques for KMDF Drivers 159
Named Device Objects 160
Security Descriptors. 160

Part 3 **Applying WDF Fundamentals**

7 **Plug and Play and Power Management 165**
Introduction to Plug and Play and Power Management 166
About Plug and Play 167
About Power States 168
About Power Policy 169
Plug and Play and Power Management Support in WDF. .170
Plug and Play and Power Management Defaults. 170
I/O Queues and Power Management. 171
Plug and Play and Power Event Callbacks 171
Idle and Wake Support (KMDF Only) 174
Power-Pageable and Non-Power-Pageable Drivers. 174
Callback Sequences for Plug and Play and Power Management 176
Device Enumeration and Startup 180
Device Power-Down and Removal 184
Surprise Removal 187
How to Implement Plug and Play and Power Management in WDF Drivers 189
Plug and Play and Power Management in Software-Only Drivers 190
UMDF Example: Plug and Play in a Software-Only Filter Driver 191
KMDF Example: Plug and Play in a Software-Only Filter Driver. 192
Framework Actions for Software-Only Drivers 194
Plug and Play and Power Management in Simple Hardware Drivers 194
Device Power-Up Initialization and Power-Down Teardown 195
Power Management for Queues in Hardware Function Drivers 196
UMDF Example: Plug and Play and Power Code in a Protocol Function Driver 197
KMDF Example: Plug and Play and Power Code in a Simple Hardware Function Driver 202
Framework Actions for a Simple Hardware Function Driver. 205
Advanced Power Management for KMDF Drivers 207

Device Power-Down Idle Support for KMDF Drivers. 208
Device Wake Support for KMDF Drivers 212
KMDF Example: Support for Device Idle and Wake 217
Framework Actions Supporting Device Idle 220
Framework Actions Supporting Device Wake 220

8 I/O Flow and Dispatching 223

Common I/O Request Types 224
Create Requests. 224
Cleanup and Close Requests 224
Read and Write Requests. 225
Device I/O Control Requests 225
Summary of I/O Request Types 227
I/O Transfer Types. 228
Buffered I/O. 229
Direct I/O 230
Neither Buffered nor Direct I/O 230
I/O Request Flow 231
I/O Request Path through the UMDF Device Stack. 234
I/O Request Path through a KMDF Driver. 236
I/O Completion Processing 238
I/O Request Flow within the Frameworks 239
Processing in the I/O Request Handler 242
I/O Request Objects 243
I/O Buffers and Memory Objects 244
Request, Memory, and Buffer Pointer Lifetimes. 252
I/O Queues 253
Queue Configuration and Request Types 254
Queues and Power Management. 257
Dispatch Type 259
Queue Control. 260
UMDF Example: Creating I/O Queues 261
KMDF Example: Creating I/O Queues 265
Retrieving Requests from a Manual Queue 267
I/O Event Callbacks. 271
File Objects for I/O 271
Automatic Forwarding of Create, Cleanup, and Close 272

I/O Event Callbacks for Create Requests 273
I/O Event Callbacks for Cleanup and Close 280
I/O Event Callbacks for Read, Write, and Device I/O Control Requests 282
Completing I/O Requests 291
Canceled and Suspended Requests 293
Request Cancellation 293
Request Suspension 296
Adaptive Time-outs in UMDF 297
Self-Managed I/O 297
Self-Managed I/O during Device Startup and Restart 300
Self-Managed I/O during Device Power-Down and Removal 300
KMDF Example: Implementing a Watchdog Timer 301

9 I/O Targets 307

About I/O Targets 308
Default I/O Targets 308
Remote I/O Targets in KMDF Drivers 309
General and Specialized I/O Targets 309
UMDF I/O Target Implementation 310
I/O Target Creation and Management 313
Default I/O Target Retrieval 313
Remote I/O Target Creation in KMDF Drivers 314
I/O Target State Management 318
I/O Request Creation 322
UMDF Example: Create a WDF I/O Request Object 322
KMDF Example: Create a WDF I/O Request Object 323
Memory Objects and Buffers for Driver-Created I/O Requests 323
Memory Object and Buffer Allocation for I/O Requests 325
UMDF Example: Create a New Memory Object with an Existing Buffer 328
KMDF Example: Create a New Memory Object and a New Buffer 330
I/O Request Formatting 330
How to Format an Unchanged Request for the Default I/O Target 331
How to Format Changed or Driver-Created Requests 331
I/O Completion Callbacks 334
How to Send an I/O Request 337
Options for Sending Requests 338
UMDF Example: Send a Request to the Default I/O Target 341

KMDF Example: Send and Forget. 343
KMDF Example: Format and Send an I/O Request to an I/O Target. 345
How to Split an I/O Request into Smaller Requests. 346
KMDF Example: Reuse an I/O Request Object 348
How to Cancel a Sent Request 351
File Handle I/O Targets in UMDF Drivers 353
USB I/O Targets. 355
About USB Devices 355
Specialized USB I/O Targets in WDF 359
How to Configure a USB I/O Target. 361
How to Send an I/O Request to a USB I/O Target 369
USB Continuous Reader in KMDF. 375
Guidelines for Sending I/O Requests. 376

10 Synchronization. 379

When Synchronization Is Required 380
Synchronized Access to Shared Data: An Example 381
Synchronization Requirements for WDF Drivers 383
WDF Synchronization Features. 384
Reference Counts and the Hierarchical Object Model 384
Serialization of Plug and Play and Power Callbacks 385
Flow Control for I/O Queues. 385
Object Presentation Lock. 386
Synchronization Scope and I/O Callback Serialization 387
Device Scope and Queue Dispatch Methods 388
Synchronization Scope in UMDF Drivers. 390
Synchronization Scope in KMDF Drivers 391
KMDF Wait Locks and Spin Locks 398
Wait Locks 398
Spin Locks 399
Synchronization of I/O Request Cancellation in KMDF Drivers. 401
Synchronized Cancellation by Using Synchronization Scope 402
Synchronized Cancellation by Tracking State in the Request Context 403
Synchronized Cancellation of Incoming Request with Driver-Created Subrequests 408
Summary and General Tips for Synchronization 409

11 Driver Tracing and Diagnosability 411

WPP Software Tracing Basics 412

Advantages of WPP Software Tracing 412

WPP Software Tracing Components 413

WPP and ETW 416

ETW in Windows Vista 416

Trace Message Functions and Macros 417

DoTraceMessage Macro 418

How to Convert Debug Print Statements to ETW 418

Message Conditions 418

Custom Trace Message Functions 419

How to Support Software Tracing in a Driver 420

Modify Sources to Run the WPP Preprocessor 420

Include the TMH File 424

Define the Control GUID and Trace Flags 425

Initialize and Clean Up Tracing 427

Instrument the Driver Code 431

Tools for Software Tracing 432

How to Run a Software Trace Session 433

Prepare the Sample 434

View a Driver Trace Log by Using TraceView. 435

View the Frameworks Trace Log by Using the Core Tracing Tools. 437

Best Practices: Design for Diagnosability. 439

12 WDF Support Objects. 441

Memory Allocation 442

Local Storage 442

Memory Objects and I/O Buffers 443

Registry Access. 447

UMDF Device Property Store. 447

KMDF Registry Objects and Methods 450

General Objects 454

UMDF Example: How to Create a General Object 454

KMDF Example: How to Create a General Object 455

KMDF Collection Objects 455

Collection Methods 456

Example: Creating and Using a Collection 457

KMDF Timer Objects 459
Timer Object Methods 459
Time Periods 461
EvtTimerFunc Callback Function 461
Example: Using a Timer Object 462
WMI Support in a KMDF Driver 464
About WMI 464
Requirements for WMI Support 465
How to Initialize WMI Support 466
WMI Instance Event Callbacks 470

13 UMDF Driver Template 475

A Description of the Skeleton Sample 475
About the Skeleton Sample 476
About the Skeleton Sample Files 476
How to Customize the Skeleton Sample Source Files 478
DLL Infrastructure 478
Basic COM Support 480
Skeleton Sample Driver Callback Object 481
The Skeleton Sample Device Callback Object 484
How to Customize the Skeleton Sample Build and Installation Support Files 486
Sources 486
Make Files 488
Exports 488
Version Resource File 488
INX File 489

Part 4 **Additional Topics for KMDF Drivers**

14 Beyond the Frameworks 497

How to Use System Services Outside the Frameworks 497
How to Use the Windows API in UMDF Drivers 497
How to Use Kernel-Mode Driver Support Routines in KMDF Drivers 500
How to Handle Requests that the Frameworks Do Not Support 503
Default Handling of Unsupported Requests 503
How to Process Unsupported Requests in KMDF Drivers 504

15 Scheduling, Thread Context, and IRQL 507

About Threads 508

Thread Scheduling. 508

Thread Context Defined 509

Thread Context for KMDF Driver Functions 510

Interrupt Request Levels. 511

Processor-specific and Thread-specific IRQLs 512

Guidelines for Running at IRQL DISPATCH_LEVEL or Higher 517

Calls to Functions that Run at a Lower IRQL 517

Thread Interruption Scenarios. 518

Thread Interruption on a Single-Processor System 518

Thread Interruption on a Multiprocessor System. 519

Testing for IRQL Problems 521

Work Items and Driver Threads. 523

About Work Items 524

KMDF Example: Use a Work Item 525

Best Practices for Managing Thread Context and IRQL in KMDF Drivers 527

16 Hardware Resources and Interrupts. 529

Hardware Resources 529

Hardware Resource Identification and Teardown. 530

Resource Lists 532

Example: How to Map Resources 534

Example: How to Unmap Resources. 537

Interrupts and Interrupt Handling 537

Interrupt Objects 538

How to Enable and Disable Interrupts 541

Post-interrupt Enable and Pre-interrupt Disable Processing 543

Interrupt Service Routines 544

Deferred Processing for Interrupts 546

Synchronized Processing at DIRQL. 547

17 Direct Memory Access 549

Basic DMA Concepts and Terminology 550

DMA Transactions and DMA Transfers. 550

Packet-Based and Common-Buffer DMA 551

Scatter/Gather Support 552

DMA-Specific Device Information 552
Device Information and DMA Driver Design 553
What Is Not a Consideration 554
Windows DMA Abstraction 555
DMA Operations and Processor Cache 556
Completion of DMA Transfers by Flushing Caches 557
Map Registers 557
System Scatter/Gather Support 560
DMA Transfer to Any Location in Physical Memory 562
Implementing DMA Drivers 564
Driver DMA Initialization 565
Transaction Initiation 569
Request Processing 572
DMA Completion Processing 575
Testing DMA Drivers 578
DMA-Specific Verification 578
The !dma Debugger Extension 579
KMDF Debugger Extensions for DMA 580
Best Practices: Do's and Don'ts for DMA Drivers 581

18 An Introduction to COM 583

Before Starting 584
UMDF Driver Structure 584
A Brief Overview of COM 586
The Contents of a COM Object 587
Objects and Interfaces 588
IUnknown 589
Reference Counting 589
Guidelines for AddRef and Release 590
GUIDs 592
VTables 592
HRESULT 593
Properties and Events 594
Active Template Library 595
Interface Definition Language Files 595
How to Use UMDF COM Objects 597
How to Start Using a COM Object 597
How to Manage a COM Object's Lifetime 600

How to Implement the DLL Infrastructure 600
DllMain 601
DllGetClassObject 602
The Class Factory 603
How to Implement UMDF Callback Objects 606
How to Implement a Class for a COM Object 607
How to Implement IUnknown 608
How to Implement UMDF Callback Objects 611

Part 5 **Building, Installing, and Testing a WDF Driver**

19 How to Build WDF Drivers. 617
General Build Considerations for Drivers. 618
UMDF Drivers—Build Issues. 618
KMDF Drivers—Build Issues 619
Introduction to Building Drivers 619
Build Environments 619
Build Utility Supporting Files 620
How to Build a Project 622
UMDF Example: Building the Fx2_Driver Sample. 625
Sources File for Fx2_Driver 625
Macros Used in the *Sources* File for Fx2_Driver 626
Makefile and Makefile.inc for Fx2_Driver. 629
How to Build Fx2_Driver. 629
KMDF Example: Building the Osrusbfx2 Sample 630
Sources File for Osrusbfx2 630
Macros Used in the *Sources* File for Osrusbfx2. 631
Makefile and Makefile.inc for Osrusbfx2 632
How to Build Osrusbfx2 633

20 How to Install WDF Drivers 635
Driver Installation Basics. 636
Key Driver Installation Tasks. 636
Installation Techniques and Tools 636
WDF Driver Installation Considerations. 637
WDF Versioning and Driver Installation. 637
How Drivers Bind to the Framework 639
WDF Co-installer Packages 640

WDF Driver Package Components. 642
How to Create an INF for a WDF Driver Package 643
Commonly Used INF Sections. 644
INF Tools. 645
INFs for Different CPU Architectures 645
INFs for WDF Drivers: The Co-installer Sections 646
Examples of WDF INFs. 648
UMDF Example: The Fx2_Driver INF 648
KMDF Example: The Osrusbfx2 INF 652
How to Sign and Distribute a Driver Package 653
Signed Catalog Files 653
How to Specify the Catalog File in the INF. 654
How to Sign Boot-Start Drivers. 654
How to Distribute the Driver Package. 655
How to Install a Driver 655
Considerations for Test Installations 655
Considerations for Release Installations 656
How to Install a Driver by Using the PnP Manager. 656
How to Install a Driver by Using DPInst or DIFxApp. 657
How to Install a Driver by Using a Custom Installation Application. 658
How to Install or Update a Driver by Using DevCon. 658
How to Update a Driver by Using Device Manager 659
How to Uninstall a Driver. 659
Driver Installation Process 659
Uninstall Actions 660
How to Troubleshoot WDF Driver Installation Problems 663
How to Use WinDbg to Debug Installation Errors. 663
Driver Installation Error Logs. 664
Common WDF Installation Errors. 665
PnP Manager Error Codes 666

21 Tools for Testing WDF Drivers. 667

Getting Started with Driver Testing 668
Choosing a Test System 668
Tools for Testing WDF Drivers: An Overview 669
About PRE*f*ast and SDV 670
Other Tools for Testing Drivers 670
Driver Verifier 677

When to Use Driver Verifier 677
How Driver Verifier Works 677
How to Run Driver Verifier 679
Driver Verifier Examples 679
How to Use Driver Verifier Information during Debugging 684
KMDF Verifier 687
When to Use KMDF Verifier 687
How KMDF Verifier Works 687
How to Enable KMDF Verifier 688
How to Use KMDF Verifier Information during Debugging 689
UMDF Verifier 690
UMDF Bug Checks 692
UMDF Error Reporting 692
Application Verifier 693
How Application Verifier Works 693
How to Use Application Verifier to Verify UMDF Drivers 693
Best Practices for Testing WDF Drivers 694
Tips for Building Drivers 694
Tips for Best Use of Tools 695
Tips for Driver Life Cycle Testing 696

22 How to Debug WDF Drivers 697

About WDF Debugging Tools 698
WinDbg 698
Other Tools 699
WPP Tracing 699
Debugging Macros and Routines 699
WinDbg Basics 699
Checked versus Free Builds 700
User Interface 700
Debugger Commands 702
Symbols and Source Code 702
Debugger Extensions 704
How to Prepare for UMDF Debugging 706
How to Enable Debugging of Driver Load and Startup Code 706
How to Start Debugging a UMDF Driver's Driver Load and Startup Code 707
How to Start Debugging a Running UMDF Driver 708

How to Track UMDF Objects and Reference Counts 709
How to Start Debugging a UMDF Driver Crash 710
How to Prepare for KMDF Debugging 711
How to Enable Kernel Debugging on the Test Computer 711
How to Prepare the Test Computer for KMDF Debugging 713
How to Start a KMDF Debugging Session 714
How to Start Debugging a KMDF Driver Crash 715
UMDF Walkthrough: Debugging the Fx2_Driver Sample. 716
Prepare to Debug Fx2_Driver 716
Start the Debug Session for Fx2_Driver. 716
Examine the OnDeviceAdd Callback Routine for Fx2_Driver. 717
Use UMDF Debugger Extensions to Examine the Device Callback Object 719
Use UMDF Debugger Extensions to Examine an I/O Request. 720
KMDF Walkthrough: Debugging the Osrusbfx2 Sample 721
Prepare for a Debug Session for Osrusbfx2 721
Start the Debug Session for Osrusbfx2 722
Examine the *EvtDriverDeviceAdd* Callback Routine. 722
Use KMDF Debugger Extensions to Examine the Device Object 723
Use KMDF Debugger Extensions to Examine an I/O Request 724
How to View Trace Messages with WinDbg. 725
How to Use WinDbg to View the KMDF Log. 726
Getting Log Information after a Bug Check. 728
Controlling the Contents of the KMDF Log 728
More Suggestions for Experimenting with WinDbg. 729

23 PRE*f*ast for Drivers 731

Introduction to PRE*f*ast 732
How PRE*f*ast Works. 732
What PRE*f*ast Can Detect. 733
How to Use PRE*f*ast 734
How to Specify the PRE*f*ast Analysis Mode 734
How to Run PRE*f*ast 734
How to Build the PRE*f*ast Examples 735
How to Display PRE*f*ast Results. 737
Examples of PRE*f*ast Results. 741
Coding Practices that Improve PRE*f*ast Results 749
Warnings that Indicate Common Causes of Noise and What to Do About Them. 750

How to Use Pragma Warning Directives to Suppress Noise 751
How to Use Annotations to Eliminate Noise 752
How to Use Annotations 753
How Annotations Improve PRE*f*ast Results 753
Where to Place Annotations in Code 755
General-Purpose Annotations 761
Input and Output Parameter Annotations 762
Annotation Modifiers 765
Buffer-Size Annotations 767
String Annotations. 772
Reserved Parameters. 774
Function Return Values 774
Driver Annotations 775
Basic Driver Annotations and Conventions 778
Conditional Annotations 780
Function Result Annotations 785
Type-Matching Annotations 786
Pointer Annotations 788
Constant and Non-Constant Parameter Annotations 789
Format String Annotations. 789
Diagnostic Annotations 790
Annotations for Functions in __try Statements 791
Memory Annotations 791
Nonmemory Resource Annotations 793
Function Type Class Annotations 800
Floating-Point Annotations 802
IRQL Annotations. 803
DO_DEVICE_INITIALIZING Annotation. 810
Annotations for Interlocked Operands 810
Examples of Annotated System Functions. 811
How to Write and Debug Annotations 814
Examples of Annotation Test Cases 814
Tips for Writing Annotation Test Cases 815
PRE*f*ast Best Practices 816
Best Practices for Using PRE*f*ast 816
Best Practices for Using Annotations 817
Example: Osrusbfx2.h with Annotations 819

24 Static Driver Verifier 823
Introduction to SDV 824
How SDV Works 825
About SDV Rules 826
How SDV Applies Rules to Driver Code 828
How to Annotate KMDF Driver Source Code for SDV 832
Function Role Type Declarations for KMDF Drivers 832
Example: Function Role Types in Sample Drivers 833
How to Run SDV 834
How to Prepare Files and Select Rules for SDV 834
How to Run a Verification 839
Experimenting with SDV 841
How to View SDV Reports 842
About SDV Defect Viewer 844
Best Practice: Check SDV Results 845
KMDF Rules for SDV 847
DDI Order Rules for KMDF 847
Device Initialization Rules for KMDF 848
Control Device Cleanup Rules for KMDF 848
Request-Completion Rules for KMDF 849
Request-Cancellation Rules for KMDF 850
Request Buffer, MDL, and Memory Rules 850
Power Policy Owner DDI Rules 853
Example: Walkthrough SDV Analysis of Fail_Driver3 855
How to Prepare to Verify Fail_Driver3 855
How to Verify Fail_Driver3 857
How to View the Results for Fail_Driver3 858
KMDF Callback Function Role Types for SDV 860
Glossary 863
Index 871

Foreword

One of the privileges—and chores—that comes with owning a core kernel component in Microsoft Windows is that I get to analyze a lot of operating system crashes that appear in that component. As the owner of the I/O manager, I have had the opportunity to debug many driver-related issues. I learned a lot from these crashes. As I debugged the crash dumps, patterns began to emerge.

To understand the problems holistically, I decided I needed a better understanding of the various device stacks—such as storage, audio, and display—and the interconnects—such as USB and 1394. So I launched what we called Driver Stack Reviews with the development leads of the device teams in the Windows Division. After numerous reviews, we concluded that our underlying driver model was too complex. We did not have the right abstractions, and we were putting too much burden on the driver developer.

The Windows Driver Model (WDM) grew organically over 14 years of development and was showing its age. Although WDM is very flexible and can support many different devices, it has a fairly low-level of abstraction. It was built for a small number of developers who had either a deep understanding of the Windows kernel or access to the kernel developers. It was not built for what is now a large pool of driver developers, who currently number in the thousands.

Too many of the rules were not well understood and were extremely difficult to describe clearly. Fundamental operating system changes like support for Plug and Play and power management were not integrated well with the Windows I/O subsystem, mainly because we wanted to be able to run both Plug and Play and non-Plug and Play drivers side by side. This meant that the operating system design pushed onto the drivers the huge burden of synchronizing Plug and Play and power events with I/O requests. The rules for synchronization are complex, difficult to understand, and not well documented. In addition, most drivers have not properly handled asynchronous I/O and I/O cancellation, even though asynchronous, cancelable I/O was designed into the operating system from the start.

Although these conclusions seemed intuitively obvious, we needed to validate them against external data. Microsoft Windows XP included a great feature called Windows Error Reporting (WER), which allows Online Crash Analysis (OCA). When Windows stops unexpectedly and displays an error message on a blue screen, the system creates a minidump of the crash, which we receive when users choose to send crash data to Microsoft. When we saw the high number of crashes, we knew that we needed to make some fundamental changes in how drivers were developed.

We also conducted a survey and held face-to-face sessions with third-party driver developers to validate our findings and present our proposal for simplifying the driver model. These discussions were eye opening. A majority of driver developers found our driver model—especially the components related to Plug and Play, power management, and cancellation of asynchronous requests—complex and difficult to use. The developers were strongly in favor of a simpler driver model. In addition, they added a few requirements that we had not considered before.

First, a simpler driver model had to work over a range of operating system platforms. Hardware vendors wanted to write and maintain a single driver for a range of operating system versions. A new driver model that worked only on the latest Windows version was not acceptable.

Second, driver developers could not be restricted to using a small set of APIs—an approach we had used in some of our device-class-specific driver models. They had to be able to escape out of the driver model to the underlying platform.

With this input, we started the work on the Windows Driver Foundation (WDF). The goal was to build a next-generation driver model that met the needs of all device classes.

For WDF, we used a different developmental methodology: We got external driver developers involved in the design right from the start by holding design reviews. As soon as we developed the specifications, we invited some developers to a roundtable discussion—the first in November 2002—so we got useful comments even before we started writing code. We sponsored an e-mail alias and discussion groups where we debated design choices. Several internal and external early adopters used our framework to write drivers and gave us great feedback. We also sought and received feedback at WinHEC and through the driver developer newsgroups.

WDF went through several iterations as it developed into what it is today. Based on the feedback we got during development, we redid our Plug and Play and power management implementation as well as our synchronization logic. In particular, the Plug and Play and power management implementation was redesigned to use state machines. This helped to make the operations explicit, so that it was easy to comprehend the relationships between I/O and Plug and Play. As more WDF drivers were developed, we discovered more rules related to Plug and Play and power management and incorporated the rules into the state machines. One of the key benefits of using WDF is that every driver automatically gets a copy of this well-tested, well-engineered Plug and Play and power management implementation.

The OCA data also indicated that we should address the problem with crashes in another, more radical way. OCA data showed that 85 percent of unexpected system stops were caused by drivers and not by core Windows kernel components. Analysis showed that drivers for many device classes—notably USB, Bluetooth, and 1394 interconnects—did not need to be in kernel mode. Moving drivers to user mode has many benefits. For example, crashes in user-mode drivers can be fully isolated and the system can recover without rebooting. The programming environment in user mode is considerably simpler than in kernel mode. Developers have access to many tools and rich languages to write their code. Debugging is much simpler. A significant advancement with WDF is that we provide the same driver model in both user mode and kernel mode.

Although driver model simplifications address many issues that cause system crashes, they do not address programmer errors like buffer overruns, uninitialized variables, incorrect usage of system routines—such as completing a request more than once—and so forth. The work at Microsoft Research (MSR) in the area of static analysis tools addressed this piece of the puzzle. MSR had developed prototypes of tools that could understand the rules of a driver model and formally analyze source code. We decided to turn two of these ideas into tools that would become part of WDF: Static Driver Verifier (SDV) and PRE*f*ast for Drivers (PFD).

With the release of Windows Vista, both the first version of WDF and our static tools became available to driver developers in the Windows Driver Kit (WDK). WDF and the static tools have laid a good foundation for our driver development platform. The initial release of Windows Vista included about 17 KMDF drivers, covering a wide variety of device classes. In user mode, both Microsoft Windows Sideshow and Windows Portable Media technologies support UMDF drivers. Microsoft will continue to build on this foundation to meet the needs of current and future device classes.

This book captures the essentials of the WDF frameworks and static tools, and it makes available for the first time a single source for all information related to WDF. The book should help any driver developer—even a novice—get up to speed quickly on WDF. You will find that WDF enables you to develop a higher quality driver in significantly less time than the older driver models.

Nar Ganapathy
Architect
Windows Device Experience Group
Microsoft Corporation

Acknowledgments

The authors acknowledge with deep gratitude the extraordinary contributions made by members of the Windows Driver Foundation team at Microsoft in providing technical information, code samples, reviews, and encouragement. Doron Holan, Narayanan Ganapathy, Praveen Rao, Eliyas Yakub, and Peter Wieland were instrumental in creating and shaping this book, which also relied on key contributions from John Richardson in creating early drafts. Donn Terry and Vlad Levin provided key guidance in developing the PRE*f*ast for Drivers and Static Driver Verifier chapters, respectively.

We salute the leadership and early development contributions to the Windows Driver Foundation by the architect on the project, Narayanan Ganapathy. Other visionary individuals who provided early support for the project were Brad Carpenter, Vince Orgovan, Bob Rinne, Rob Short, and Mike Tricker.

Johan Marien, program manager during the development phase for WDF, launched the ongoing effort to discuss driver model directions with the Windows driver developer community. We want to acknowledge the significant contributions from Peter Viscarola and Open Systems Resources, Inc. (OSR), especially for their fervent evangelism and their work on the OSR USB Fx2 Learning Kit, upon which several of the WDF samples are based. Valuable review comments and great ideas were also contributed by several of the driver development community's "Most Valued Professionals" and leading Windows driver developers: Don Burn, Trevor Goveas (Agere), Bill McKenzie, Tim Roberts (Providenza & Boekelheide), Mark Roddy (Hollis Technology Solutions), Eric Tissot-Dupont (Logitech), and Ray Trent (Synaptics).

A project as extensive as WDF would not be possible without the dedicated product development teams working to turn a vision into a product.

WDF would not exist without the extraordinarily talented design and development team. Significant contributors were Robin Callendar, Joe Dai, Doron Holan, Vishal Manan, Adrian Oney, Jake Oshins, Ray Patrick, Guruprakash Rao, Abhishek Ram, Praveen Rao, John Richardson, Mukund Sankaranarayan, Erick Smith, Peter Wieland, and Eliyas Yakub.

Static analysis and verification tools are a key part of WDF. Tom Ball and Sriram Rajamani invented the SLAM verification engine and, supported by the WDF group, made a successful presentation to Bill Gates on a state-of-art driver verification technology based on SLAM. These events led to a launch of the Static Driver Verifier project in the WDF group. Further important contributions to the static tools were made by Vlad Levin, Donn Terry, Ella Bounimova, Byron Cook, Jakob Lichtenberg, and Con McGarvey.

The WDF test team ensured the quality of the frameworks. The leaders for test development were Quetzel Bradley for WDF and Static Tools, Ravi Gollapudi for KMDF, and Abdullah Ustuner for UMDF and Static Tools. KMDF Test contributors were Aruna Banda, Bob Kjelgaard, Kumar Rajeev, and Willem van der Hoeven. UMDF test contributors were Shefali Gulati, Shyamal Varma, Jimmy Chen, Patrick Maninger, and James Moe. Static Tools test contributors were Jon Hagen, Onur Ozyer, and John Henry.

Documentation is critical for all products and especially for developer tools. Richard Brown, Dave Hagen, John Jackson, and Adam Wilson contributed to the WDF documentation.

We appreciate Microsoft management for believing and investing in the WDF project and the program managers for helping to identify and remove road blocks to bring the Windows Driver Foundation to the driver community. The managers on the project were Fran Dougherty, Stu Farnham, and Harish Naidu. The program managers on the project were Johan Marien, Jeffrey Copeland, Murtuza Naguthanawala, Bohus Ondrusek, and Teresa Stone.

Many Microsoft teams contributed to the WDF initiative by becoming early adopters of WDF and providing feedback to the development team:

- Microsoft Hardware: Vadim Dmitriev
- Tablet PC: Mikki Durojaiye
- Windows Client Technologies for Emerging Markets: Zhangwei Xu
- Windows I/O Manager: John Lee, Paul Sliwowicz
- Windows Media Player/Windows Portable Devices: Oren Rosenbloom, Vlad Sadovsky, Byron Changuion, Cooper Partin, E-Zu Wu, Jim Bovee, John Felkins, Blake Manders
- Windows SideShow: Dan Polivy
- Windows Universal Audio Architecture - High Definition Audio: Hakon Strande, Frank Berreth, Cheng-mean Liu
- Windows Virtualization: Jake Oshins, Benjamin Leis
- WinUSB: Randy Aull
- Xbox 360 Controller: Matt Coill

– from the authors, *Developing Drivers with the Microsoft Windows Driver Foundation*:
Penny Orwick, Guy Smith, Carol Buchmiller, Annie Pearson, Gwen Heib, and the Windows Hardware Developer Central (WHDC) team at Microsoft.

Part 1
Getting Started with WDF

In this part:

Chapter 1: Introduction to WDF 3

Chapter 2: Windows Driver Fundamentals 23

Chapter 3: WDF Fundamentals 51

Chapter 1

Introduction to WDF

In this chapter:

About This Book 4
Conventions Used in This Book 9
Getting Started with Driver Development 11
Key Information Sources 20

For many years, software developers have found implementing drivers for Microsoft Windows operating systems to be a challenging task. Until recently, the steep learning curve of the Windows Driver Model (WDM) has limited driver development to a relatively small group of specialized developers.

Windows Driver Foundation (WDF) provides a driver model that makes it easier to learn and easier to implement robust Windows drivers. WDF largely supersedes WDM and is designed to enable developers to focus on the requirements of their hardware rather than the complexities of the operating system. WDF also enhances system stability by supporting the ability to create drivers that run in user mode for several important device categories that previously required kernel-mode drivers.

With the WDF programming model, a developer can quickly implement a basic but functional driver, with WDF handling most of the event processing. The developer can then incrementally expand the scope of handled events until the driver is fully functional.

This book is designed to introduce WDF to anyone who is interested in developing Windows drivers, including software programmers with no previous driver development experience. We wrote this book in partnership with the WDF development team at Microsoft, who designed the architecture, built the frameworks, and developed the sample drivers to guide programmers. This book starts with a high-level discussion of the WDF architecture and programming model, but most of it is designed to provide a practical, sample-oriented introduction to the WDF frameworks for developing Windows drivers.

What Criteria Did We Use for Creating WDF APIs?

When we were designing the WDF architecture, the first and ongoing criterion we used for adding an API was if it was a "thin book" or "thick book" API.

"Thin book" meant it was both a simple API that most driver developers would use and that its use would be easy to determine based on its signature alone (such as **WdfDeviceInitSetExclusive**) and the documentation for it could be included in a short document like a white paper (hence, "thin").

"Thick book" meant that the API was going to be difficult to understand or rarely used, requiring the developer to delve deep into the documentation to learn how to use it.

This criterion really drove our decisions in terms of what types of APIs we exposed ("thin book") and what the frameworks did "under the hood" on behalf of the driver. Hopefully, we made the right decisions and you'll enjoy this "thick book" on how to delve deep into the frameworks.

–Doron Holan, Windows Driver Foundation Team, Microsoft

About This Book

This book is structured to provide you with the basic information you need to develop a WDF driver and assumes that you have no prior experience with driver development. The book starts with how a driver functions in the Windows operating system environment, and then builds on this information to describe how to use WDF to implement drivers.

Who Should Read This Book

This book is intended for developers with a solid foundation in programming with the C or C++ language who are interested in implementing Windows drivers, including:

- **Driver developers who are interested in learning how to implement drivers with WDF.** If you are an experienced driver developer, you should find it easy to adapt to the new model. However, WDF user-mode drivers are COM based, so this book provides a basic introduction to COM programming for those who need it.
- **Application developers who want to get started in driver development.** WDF provides a much easier learning curve than earlier driver models, but applications differ in many ways from drivers, especially kernel-mode drivers. This book presents basic background information to help you understand the structure and operation of Windows drivers. This background information is then used in the discussions of driver development throughout this book.

- **Hardware engineers who need to understand how drivers interact with devices.** Engineers who must build drivers for their prototype hardware will find the WDF model especially useful because of its rapid prototyping and easier learning curve.

Under WDF, developing user-mode drivers requires an understanding of C++, whereas kernel-mode drivers are almost always written in C. If you are unfamiliar with either language, you should consult any of the many books on these languages. By building on your knowledge of C and C++ and your familiarity with Windows programming, this book provides the concepts, guidelines, programming examples, and tips to get you started with WDF drivers.

About Part 1: Getting Started with WDF

The first part of this book provides an introduction to tools and resources, basic Windows operating system and driver concepts, and an overview of WDF.

Chapter 1, "Introduction to WDF" (this chapter) This chapter provides an orientation to this book and the tools it discusses. To begin:

- Review this chapter to understand the structure of this book and the conventions used in text and in code examples.
- Review the requirements and install the Windows Driver Kit (WDK) and debugging tools as described in "Getting Started with Driver Development" later in this chapter, so that you can follow the samples and try activities in real time.

Chapter 2, "Windows Driver Fundamentals" This chapter covers the background material that you'll need to understand driver development. It is especially intended for application programmers who are not familiar with drivers or kernel-mode programming. Read this chapter for a quick primer on Windows architecture, the Windows I/O model, user mode versus kernel mode, Plug and Play, interrupts, memory management, threads and synchronization, and other key concepts.

The topics in Chapter 2 are all revisited in depth later in the book in the context of WDF driver implementation.

Chapter 3, "WDF Fundamentals" WDF includes three primary components: the user-mode driver framework (UMDF), the kernel-mode driver framework (KMDF), and a suite of verification and testing tools. This chapter provides a conceptual tour of the WDF architecture and introduces the two frameworks. This chapter also introduces basic concepts related to WDF, such as the WDF object model, I/O model, and how WDF manages Plug and Play and power events.

About Part 2: Exploring the Frameworks

Although the two WDF frameworks are similar in many ways, they are not identical; each has its strengths and limitations. In Part 2, you'll find a detailed tour of the frameworks.

Chapter 4, "Overview of the Driver Frameworks" This chapter describes the two frameworks, including their related runtime components:

- UMDF supports the creation of simple, robust Plug and Play drivers for several device categories, especially for consumer-oriented devices such as portable media players and cell phones.
- KMDF supports the creation of drivers that must run in kernel mode because they support direct memory access (DMA), handle interrupts, have strict timing loops, are clients of other kernel-mode drivers, or require kernel-mode resources such as the nonpaged pool.

This chapter also provides guidelines to help you choose which framework to use to implement a driver for a particular device.

Chapter 5, "WDF Object Model" WDF supports an object-oriented, event-driven programming model in which:

- The basic building blocks of drivers are objects. The object models for UMDF and KMDF are similar, although the implementation details differ.
- Each object has an associated set of events. The frameworks provide default handling for most events. Drivers handle only those events that are relevant to their device and let the frameworks handle the remainder.

A WDF driver interacts with these objects through consistent, well-defined programming interfaces. This chapter provides details about the object model as a foundation for understanding what you must implement in your driver and what the frameworks handle.

Chapter 6, "Driver Structure and Initialization" To help you get started with the specifics of Windows driver development, this chapter explores the structure and required components of WDF drivers. It also explores common aspects of UMDF and KMDF drivers: driver objects, device objects, driver entry points, and callbacks. You'll learn about driver entry routines, initialization, and device object creation.

About Part 3: Applying WDF Fundamentals

After you have a solid conceptual understanding of the architecture and components of the WDF frameworks, it's time to dive into the details of Windows drivers that will absorb much of your time and energy during development. This part of the book explores a number of important concepts and practices.

Chapter 7, "Plug and Play and Power Management" Plug and Play is a combination of hardware and software that enables a computer to recognize and support hardware configuration changes with little or no user intervention. Windows also supports a power management architecture that provides a comprehensive policy for managing system and device power.

These two Windows capabilities have proved difficult for driver developers who used earlier driver models. This chapter explores how these features work in WDF drivers and shows how the frameworks substantially reduce and simplify the code that is required to support Plug and Play and power management.

Chapter 8, "I/O Flow and Dispatching" A Windows driver receives I/O requests from applications, services them, and returns information to the application. This chapter provides a general description of I/O flow in WDF drivers, describes the types of I/O requests that drivers might be asked to handle and how to create queues to handle them, and focuses in detail on some commonly used request types.

Chapter 9, "I/O Targets" Drivers can satisfy and complete some I/O requests, but they must pass other requests to lower components of their device stack or to other device stacks. Drivers can also issue I/O requests. A WDF driver uses an I/O target to send an I/O request to another driver—whether that driver is in the same device stack or a different one. This chapter explores the details of creating I/O targets and sending I/O requests, including information on specialized I/O targets for USB devices.

Chapter 10, "Synchronization" Windows is a preemptive, multitasking operating system, which means that different threads can try to concurrently access shared data structures or resources and that multiple driver routines can run concurrently. To ensure data integrity and to avoid race conditions, all drivers must synchronize access to shared data structures and resources. This chapter discusses when synchronization is required and then explores the synchronization and concurrency features that the frameworks provide.

Chapter 11, "Driver Tracing and Diagnosability" Software tracing provides a low-overhead way to analyze your driver's behavior. This chapter discusses how to use Windows Software Trace Preprocessor (WPP) to instrument a WDF driver to help analyze your driver's behavior and fix its problems. The emphasis in this chapter is on best practices that help you design your driver for diagnosability.

Chapter 12, "WDF Support Objects" All drivers use the device, driver, and I/O queue objects described in earlier chapters. The frameworks also define additional objects that represent less-common driver abstractions. This chapter describes some of the other objects that you'll use to implement WDF drivers and describes techniques for allocating memory, reading and writing to the registry, using timers and collections, and supporting Windows Management Instrumentation (WMI) in KMDF drivers.

Chapter 13, "UMDF Driver Template" The Skeleton sample driver contains the minimum amount of code that is required in a UMDF driver. You can use it as a starting point from which to build drivers for actual hardware. This chapter explains how the Skeleton driver demonstrates the minimal required features and best practices for a UMDF driver, and then it describes how to use the sample as a starting point for implementing a full-featured driver.

About Part 4: Digging Deeper: More Topics for WDF Drivers

A kernel-mode driver is, in effect, part of the Windows operating system and consequently must manage additional complications that do not apply to user-mode drivers. KMDF drivers might be required to deal with the subtleties of hardware interrupts and direct memory access. For UMDF drivers, you need to understand how to use and implement COM objects. These deeper subjects are explored in this part of the book.

Chapter 14, "Beyond the Frameworks" Although the frameworks provide most of the features that drivers use most of the time, there are some exceptions when drivers require services that the frameworks do not support. This chapter describes how to use system services that fall outside the frameworks. For example:

- UMDF drivers can use many of the functions in the Windows API.
- KMDF drivers can use kernel-mode system functions, including functions that manipulate the WDM objects underlying the WDF objects.

Chapter 15, "Scheduling, Thread Context, and IRQL" Thread scheduling, thread context, and the current interrupt request level (IRQL) for each processor affect how kernel-mode drivers work. This chapter explores the concepts and best practices that you must master to avoid problems related to interrupts, preemption, and IRQL in KMDF drivers.

Chapter 16, "Hardware Resources and Interrupts" If your device hardware generates interrupts, your kernel-mode driver must service those interrupts. To service a hardware interrupt, a KMDF driver must create an interrupt object, enable and disable the interrupt, and respond appropriately when an interrupt occurs. This chapter discusses how to service interrupts with KMDF and provides guidelines and best practices.

Chapter 17, "Direct Memory Access" DMA is a high-performance technique for transferring data directly to and from memory for DMA-capable devices. DMA can support higher data rates than other approaches, along with lower overall system CPU usage. KMDF transparently handles much of the work that is required to implement DMA in a driver. This chapter describes basic DMA concepts and how to implement DMA in a KMDF driver.

Chapter 18, "An Introduction to COM" To create a UMDF driver, you must use a number of COM objects that belong to the UMDF runtime and also implement a number of COM-based callback objects. This chapter provides a basic introduction to using and implementing COM objects, as required by UMDF.

About Part 5: Building, Installing, and Testing a WDF Driver

Drivers must be built, tested, debugged, and installed by using a set of tools and techniques designed specifically for driver development. In addition to the standard tools, WDF includes a set of verification, testing, and debugging tools to make it easier to produce robust WDF drivers.

Chapter 19, "How to Build WDF Drivers" Window drivers are built with the WDK build utility, Build.exe. This command-line utility is the tool that Microsoft uses to build Windows. This chapter shows how to set up your build environment and describes how to build UMDF and KMDF drivers.

Chapter 20, "How to Install WDF Drivers" Installing a driver is quite different from installing an application. This chapter explores the tools and processes for installing drivers, including the use of tools such as DevCon and Device Manager. It also closely examines INF issues for WDF drivers, and points to resources for code-signing drivers.

Chapter 21, "Tools for Testing WDF Drivers" Thorough testing throughout all phases of development is essential to create a robust, high-quality driver. WDF includes built-in runtime verification for both frameworks, adding to Driver Verifier and other general-purpose driver-testing tools that are provided in the WDK. This chapter provides a brief introduction to the tools and best practices to test and verify WDF drivers.

Chapter 22, "How to Debug WDF Drivers" WinDbg is the debugger of choice for both UMDF and KMDF drivers. This debugger works in either user mode or kernel mode and supports a number of extensions that simplify debugging WDF-specific issues. This chapter introduces driver debugging and provides a basic overview of how to use WinDbg to debug WDF drivers.

Chapter 23, "PRE*f*ast for Drivers" PRE*f*ast for Drivers is a static source code analysis tool for driver development. This tool runs at compile time and reports a variety of coding errors that the compiler and runtime testing are unable to detect in drivers. This chapter explores how to use PRE*f*ast effectively in your driver development process, including the use of source code annotations in your driver so PRE*f*ast can perform a deeper analysis of your code.

Chapter 24, "Static Driver Verifier" Static Driver Verifier (SDV) is a static verification tool for kernel-mode drivers that emulates the operating system's path through the driver and symbolically executes the source code. This tool has built-in knowledge about Windows internals and about how drivers should use Windows interfaces. This chapter describes the use of SDV as a recommended best practice for driver development. It includes details about KMDF rules for SDV, first introduced in 2007.

Glossary The glossary contains a list of the terms and acronyms used in this book. A comprehensive glossary of driver development terms is provided in the WDK.

Conventions Used in This Book

Most of the conventions in this book are the same as those in the WDK documentation and in other Microsoft products for programmers. In addition, this book provides references to help you find the samples, documentation topics, white papers, tools, or other information in each specific discussion. Table 1.1 summarizes the typographical and other conventions used in this book.

Table 1-1 Documentation Conventions

This convention …	Indicates …
bold	System-supplied or system-defined functions and support routines, structure members, enumerators, and registry key names. These items appear in the system header files that are included with the WDK exactly as shown in this documentation. For example, **WdfCreateDevice** is a system-supplied function that supports kernel-mode WDF drivers.
italic	Placeholder function names, formal parameters, or any other text that is meant for you to replace in your own code. Portions of registry paths or INF entries in italics are placeholders, to be replaced with driver- or device-specific text. For example: ● *EvtDriverDeviceAdd* is a placeholder name for a callback function that the driver defines. ● `#pragma warning(disable:`*`WarningNumber`*`)` uses a placeholder for a numeric value that would appear in an actual warning.
`Monospace`	Code examples, such as: `hwInitData.DeviceIdLength=4`
UPPERCASE	Constant identifiers, data type names, bitwise operators, and system-supplied macros. Uppercase identifiers must be typed exactly as shown. For example, WDF_DRIVER_CONFIG is a system-defined structure.
Filename.txt	The name of a file. This book shows file names in upper and lower case type for better readability. File names are not case-sensitive.
%wdk%	The root installation directory for the WDK—typically, C:\WinDDK*BuildNumber*.
%windir%	The root installation directory for the Windows operating system—typically, C:\Windows.
x86, x64, Itanium	References to the different CPU architectures that run Windows, specifically: ● x86 for 32-bit processors that run the Intel instruction set, ● x64 for 64-bit processors such as AMD64, and ● Itanium for Intel Itanium processors.

Table 1-1 Documentation Conventions

This convention ...	Indicates ...
\<i386 | amd64 | ia64>	Alternative subdirectories in a WDK folder that contain files for the different hardware platforms, specifically: ● i386 for x86 versions of Windows, ● amd64 for x64 versions of Windows, and ● ia64 for 64-bit Windows on the Itanium platform. For example: %wdk%\tools\acpi\i386\.

Finding Resources for Each Chapter

Each chapter begins with a list of the samples, documentation, and tools you need to follow along on your personal workstation. The following shows an example.

For this chapter, you need ...	From ...
Tools and files	
Build.exe	%wdk%\bin\<amd64 | ia64 | i386>
Sample drivers	
Fx2_Driver	%wdk%\src\umdf\usb\fx_2driver
Osrusbfx2	%wdk%\src\kmdf\Osrusbfx2
WDK documentation	
UMDF Objects and Interfaces	http://go.microsoft.com/fwlink/?LinkId=79583
Other	
"Developing Drivers with WDF: News and Updates" on the WHDC Web site	http://go.microsoft.com/fwlink/?LinkId=80911

The WDK topics listed in each chapter's resources include links to the online version of the documentation on MSDN. Other links are for white papers and Web-based resources. You can find all these references as convenient hyperlinks at "Developing Drivers with WDF: News and Updates" on the WHDC Web site.

Getting Started with Driver Development

To follow the examples in this book and to use the WDF frameworks, you must install the current Windows Driver Kit (WDK). The WDK contains most of the resources you need to develop drivers, such as tools, documentation, and libraries. The following sections provide some guidelines and tips for installing software and finding the samples and tools discussed in this book.

> **Important** Always use the most recent version of the WDK. This book assumes that you are using WDK Build 6000 or a later version. If you are already developing Windows drivers, you need to install Build 6000 or a later version to obtain the components discussed in this book. Static Driver Verifier for KMDF drivers and PRE*f*ast driver-specific annotations require the WDK version provided with the Beta 3 release of Microsoft Windows Server Code Name "Longhorn," or a later version.

System Requirements for Driver Development

You can develop and build KMDF drivers for Windows 2000 or later versions of Windows. You can build UMDF drivers for Windows XP or a later version of Windows. You can use any recent version of Windows to build your drivers. To target a driver for a particular version and CPU architecture, you specify the appropriate build environment configuration when you use the Build utility.

However, you should plan to install, test, and debug your driver on a system that is running the target version of Windows, with hardware that is the same as or similar to the hardware on your customers' systems.

> **Important** Always use the most recent version of the Debugging Tools for Windows, as described later in this chapter.

To debug KMDF drivers, you need two computers: one to host the debugger and the other to host the driver you want to debug. Kernel-mode driver bugs commonly cause system crashes and can corrupt the file system, causing loss of data—so it's necessary to have the debugger and driver on separate computers.

To debug UMDF drivers, you can run the debugger and driver on the same computer or run the two on separate systems. You can also use a single computer with different versions of Windows on separate partitions for many of these tasks. However, a common practice is to have a development computer plus at least one additional computer dedicated to testing and debugging.

We recommend the following hardware for a developer's work configuration:

- **A computer with a multicore processor or multiple processors.** At a minimum, your processor should support hyperthreading. If you test your drivers only on a uniprocessor system, you might not detect certain types of bugs, such as race conditions. In addition, the Windows Logo Program requires that all submitted drivers pass compatibility tests for multiprocessor systems.
- **A 64-bit computer that runs x64 versions of Windows.** Certain types of critical errors can be detected only on 64-bit systems. The Windows Logo Program requires that your driver support both 32- and 64-bit systems.

> **Tip** If you develop and test WDM drivers by using WDK Build 6000 or later, you can use the same build environments to develop drivers with WDF.

How to Obtain and Install the WDK

The WDK, which contains the Driver Development Kit (DDK), is the driver developer's primary resource. The WDK contains the bulk of what you need to develop drivers, as described in the following sections.

The WDK supports driver development for the Microsoft Windows NT family of operating systems, starting with Windows 2000. The WDK is released periodically, and the version is typically associated with a particular Windows release. However, that's just a convenient way of indicating the release date. In fact, each WDK release supports building drivers for all hardware platforms and all versions of Windows that Microsoft supports under the life cycle support policy. In 2007, the WDK supports building drivers for Windows 2000, Windows XP, Windows Server 2003, Windows Vista, and Windows Server "Longhorn."

You should always use the latest version of the WDK. This practice guarantees that you have current documentation and tools, including all updates and fixes made since the previous release.

To obtain the WDK

1. See the WHDC Web site at http://www.microsoft.com/whdc/ for information about obtaining the current WDK.

   New versions are released in conjunction with associated product releases, such as Windows beta and RTM releases and major developer events such as WinHEC. The WHDC Web site also provides information about which versions of Windows are supported by the current WDK.

2. If you download the ISO for the WDK, burn a CD or DVD to create the installation media.

To install the WDK

1. Insert the WDK installation media to run the installation application.
2. Read the WDK Release Notes to check for any installation issues.
3. Install required prerequisites.

   The installation application checks your system and enables buttons under **WDK Prerequisite Setup Packages** for any features that are not present. You might not need all of these features, but you probably want the .NET Framework, Version 2.0, and Microsoft Document Explorer to view the documentation.

4. Under **WDK Setup Package Features**, click **Install**.

By default, the WDK installs on your C drive in a root folder named WinDDK *BuildNumber*. If you install multiple WDK versions, the WDK installation wizard places each WDK version under its own build number.

> **Tip** Because you can choose to install the WDK on other drives the %wdk% environment variable is used in this book to refer to the WDK root folder.

WDK Libraries

The WDK includes static libraries (LIBs), dynamic-link libraries (DLLs), and the WDF libraries that you need for developing WDF drivers:

- **Redistributable co-installer DLLs** The WDK contains redistributable co-installers for UMDF and KMDF. The co-installers are used during driver installation to install the associated framework runtime support on a user's computer, if the runtime is not already present. Drivers dynamically bind with the frameworks that the co-installers install. To make the co-installers easier to identify, their names include the WDF version number. There are separate versions of the co-installers for each supported CPU architecture.

  Chapter 20, "How to Install WDF Drivers," describes the co-installers.

- **Debugger extensions** The WDF debugger extensions are specialized, WDF-specific commands that run in the context of the WinDbg debugger. The debugger extensions are packaged in two DLLs: WudfExt.dll contains the UMDF extensions, and WdfKd.dll contains the KMDF extensions. Separate versions of the debugger extensions are provided for each supported CPU architecture.

  Chapter 22, "How to Debug WDF Drivers," describes WinDbg and the extensions.

- **Libraries** The WDK contains a number of static libraries. KMDF drivers bind statically with WdfDriverEntry.lib and WdfLdr.lib. You can choose to implement UMDF drivers by using the Active Template Library (ATL), a C++ template library that is designed to simplify the process of implementing COM objects.

  KMDF libraries are located under %wdk%\lib\wdf\kmdf. ATL libraries are located under %wdk%\lib\atl.

WDK Documentation

This book is only an introduction to WDF development. For all of the necessary information to develop a fully functional device driver, see the WDK documentation. That documentation provides detailed reference pages for every function exposed in the WDF device driver interface (DDI). The WDK documentation also contains conceptual material, design and implementation guidelines, and documentation for the WDK tools.

To run the WDK documentation from the Start menu

- On the taskbar, click **Start**, and then click:

  All Programs > **Windows Driver Kits** > *BuildNumber* > **Help** > **WDK Documentation**

You can view the WDK documentation online as part of the MSDN Library—online at http://msdn.microsoft.com. Locate the Microsoft Win32 and COM Development node in the MSDN Library table of contents, and then find Windows Driver Kit. The online version of the WDK documentation is updated quarterly.

WDK Tools

The WDK contains a substantial set of tools for development and testing. Most are command-line tools that run in a command window, but some are conventional Windows applications with graphical user interfaces. Some tools for testing include the following:

- **Tracing** Chapter 11, "Driver Tracing and Diagnosability," describes how to use WPP tracing and the associated WDK tools to trace execution and assist in debugging.
- **Driver Verifier and other runtime tools** Chapter 21, "Tools for Testing WDF Drivers," discusses the runtime verification tools for testing drivers.
- **Static verifiers** Chapter 23, "PRE*f*ast for Drivers," and Chapter 24, "Static Driver Verifier," provide details about using static verifier tools.

Note To run some of the tools in Windows Vista, you must specify that you want the tool to run with elevated privileges, even if you already have administrative privileges. See "User Account Control" on MSDN for details—online at http://go.microsoft.com/fwlink/?LinkId=80621.

To run an application on Windows Vista with elevated privileges

1. On the taskbar, click **Start**, right-click the application, and then click **Run as administrator**.

   If you already have administrative privileges, Windows Vista displays a **User Account Control** dialog box asking for permission to proceed.
2. To run the application, click **Continue**.
3. If you do not have administrative privileges, Windows Vista asks for administrative credentials.

To open a command window with elevated privileges

1. On the taskbar, click **Start**, right-click **Command Window**, and then click **Run as administrator**.
2. Click **Continue** and provide credentials, if prompted.

   Any application that you run from that command window, including any Windows-based application, also has elevated privileges.

WDK Samples

The WDK includes an extensive set of samples that show common types of drivers you might develop. These samples contain well-designed working code and are extensively commented.

The WDF samples are installed with the WDK in %wdk%\src\kmdf and %wdk%\src\umdf. Each sample is installed in its own folder with the same name as the driver.

> **Tip** Before you build or modify any WDK sample, copy the files to another directory and then work with the copies. This preserves the sample in its original form in case you need it.

About UMDF Samples

If you are a new WDF developer, we recommend that you examine the Skeleton and Fx2_Driver samples, which are used throughout this book:

- **Skeleton driver (UMDF Template)** This basic driver does little more than load and start successfully.

  The Skeleton sample driver is simple so that you can use it to learn the basics of how a UMDF driver works. It's also a good starting point for implementing a functional driver, because most UMDF drivers can use most of the code in the Skeleton sample driver with little or no modification. You can add code incrementally to handle the requirements of your particular device.

  The Skeleton sample driver is installed with the WDK in the %wdk%\src\umdf\skeleton folder.

- **Fx2_Driver** This sample is a USB driver that was specifically designed for learning purposes.

  The Fx2_Driver works with the OSR USB FX2 Learning Kit device from Open System Resources (OSR), as described later in this chapter. Fx2_Driver is simple because the device itself is not very complicated. However, the driver works with a real device and demonstrates a range of basic UMDF features, including how to handle read, write, and device I/O control requests.

  Code from the Fx2_Driver sample driver appears throughout this book in examples to show how UMDF drivers work. Other samples are also included—in this book and in the WDK—to illustrate features that Fx2_Driver does not support.

  Fx2_Driver is located under the %wdk%\src\umdf\usb\fx2_driver folder. Each subfolder contains a variant of the driver. The driver in the Step1 folder is a minimal implementation. The Step2 through Step5 folders contain increasingly full-featured versions of the driver. The Final folder contains the complete driver and is the sample that is used in this book. The \Exe subfolder contains source code for a simple console application that can be used to operate the device and exercise the capabilities of the driver.

About KMDF Samples

The WDK provides many KMDF samples. If you are a new WDF developer, we recommend that you examine the Toaster and Osrusbfx2 samples, which are used throughout this book. You can examine other samples for more information about other features that these samples do not use, such as DMA or interrupt handling:

- **Toaster** This software driver simulates the behavior of real devices in Windows.

  The Toaster driver is not quite as basic as the Skeleton driver, but nevertheless serves as a useful starting point for understanding KMDF drivers. The Toaster sample is made up of a collection of related drivers, including bus, filter, and function drivers.

  The Toaster sample is located in the %wdk%\src\kmdf\toaster folder. Several subfolders contain different related drivers. The best place to start is with the function driver located under %wdk%\src\kmdf\toaster\func.

  Two drivers in that subfolder contain a different version of the driver: a basic one named Simple, and a full-featured version named Featured. The %wdk%\src\kmdf\toaster\exe folder contains the source code for several test applications that you can use to exercise the driver's capabilities.

- **Osrusbfx2** This sample is a USB driver built for kernel mode.

  Code from the Osrusbfx2 sample driver appears throughout this book in examples to show how KMDF drivers work. This sample does not demonstrate all of the features of KMDF. The %wdk%\src\kmdf folder in the WDK contains other samples that illustrate features that Osrusbfx2 does not support.

  This sample is the kernel-mode equivalent of Fx2_Driver. It has almost the same capabilities as Fx2_Driver, and the structure and code are quite similar.

  Osrusbfx2 is located in %wdk%\src\kmdf\osrusbfx2, which includes six subfolders. The Step1 through Step5 and the Final folders contain successively more sophisticated versions of the driver. Each version has similar capabilities to the corresponding Fx2_Driver version. The \Exe subfolder contains the source code for a simple test application.

> **Tip** Use the Osrusbfx2 test application with both the OsrUsbFx2 and Fx2_Driver samples. It provides access to all the features on the device.

How to Obtain Checked Builds of Windows

The system Microsoft uses to build Windows produces two separate builds of the operating system: free and checked. We recommend that you test and debug on both types of builds:

- **Checked builds** Checked builds contain detailed debugging information and enable certain types of debugging-related code such as ASSERT macros.

Checked builds are similar to the debug builds that are used in application development. They are typically slower than free builds. Because some compiler optimizations are disabled, the disassembled machine instructions and trace messages are more easily understood.

- **Free builds** Free builds lack detailed debugging information and are fully optimized.

  These builds are similar to the release builds that are used in application development. Retail versions of Windows are all free builds because they have the best performance and smallest memory footprint.

We recommend that you use checked builds of Windows for testing and debugging. Developers typically switch to free builds of Windows for performance tuning late in the development cycle, after most of the bugs have been eliminated.

You can obtain checked builds on MSDN Subscriber CDs or from the Subscription area on MSDN. See "Using Checked Builds of Windows" on the WHDC Web site—online at http://go.microsoft.com/fwlink/?LinkId=79304.

How to Obtain Debugging Tools

The Debugging Tools for Windows package is provided at no charge on the WHDC Web site and is included in the WDK.

To obtain Debugging Tools for Windows from WHDC

1. Go to the "Debugging Tools for Windows" page on the WHDC Web site at http://go.microsoft.com/fwlink/?LinkId=80065.
2. Locate the **Install** page for the 32-bit version and follow the instructions for downloading the package.
3. Locate the page for the 64-bit version and repeat the steps to download the 64-bit package.

Check the Debugging Tools for Windows documentation for information about where the package is installed.

In general, if the debugger is running on a 32-bit system, you must use the 32-bit package. If the debugger is running on an x64 system and the target computer is running Windows XP or later, you can use either the 32-bit or the 64-bit package. See "Choosing Between the 32-bit and 64-bit Packages" in the Debugging Tools for Windows documentation.

The Debugging Tools for Windows package includes the following components:

- **Debuggers** WinDbg is a graphical debugging tool that we recommend for both UMDF and KMDF drivers. However, developers who prefer a command-line tool can also use KD, a console application with the same capabilities as WinDbg.

- **A collection of related debugging tools** The Debugging Tools for Windows package also includes other tools to support debugging. For example, Tlist is a command-line tool that displays information about running processes and is useful for debugging UMDF drivers. The DBH tool can be used to look at symbols while debugging.
- **Debugging documentation** The Debugging Tools for Windows Help file contains instructions on how to use the debugging tools and a complete reference for the standard debugger commands and extensions.

When debugging a driver—especially a kernel-mode driver—you typically need the symbol files for the version of Windows under which the driver is running. Symbol files for all versions of Windows are available from Microsoft.com. For kernel debugging, you must install the symbols on the computer that hosts the debugger, not on the test computer.

Symbols for the hardware abstraction layer (HAL) and Windows kernel (KRNL) are installed with the WDK at %wdk%\debug.

To obtain up-to-date symbols, we recommend that you connect the debugger to the Microsoft symbols server, which automatically downloads the correct symbols.

To obtain Windows symbols

- Follow the instructions in the WinDbg help file to connect to the Microsoft symbols server, from which WinDbg can automatically download the correct symbols—online at http://msdl.microsoft.com/download/symbols.

  –Or–

  Download the current packages from the WHDC Web site at http://go.microsoft.com/fwlink/?LinkId=79331.

Chapter 22, "How to Debug WDF Drivers," discusses how to use symbols when debugging a driver, including how to use the Microsoft symbols server.

See "Debugging Tools and Symbols—Resources" on the WHDC Web site for a list of training companies and other resources for learning about debugging Windows drivers—online at http://go.microsoft.com/fwlink/?LinkId=79332.

How to Obtain OSR Learning Devices

The best way to learn how to develop device drivers is with an actual device. However, commercial devices are frequently complex, making it difficult to get started. In addition, the specifications you need to implement such a driver are often proprietary and can be difficult to obtain if you do not work for the manufacturer.

To solve this problem for beginning driver developers, OSR has created several learning kits with devices that are specifically designed for learning driver development. They are simple enough that you can focus on developing basic skills.

Physically, the kits are circuit boards that either plug in to the PCI bus or attach to the computer with a USB cable. The boards provide visual feedback on their operation, so that you can easily see what is happening. For example, the USB device has an LED panel that can be programmed to display alphanumeric characters, as shown in Figure 1-1. This book uses Microsoft sample drivers that work with the OSR USB Fx2 device.

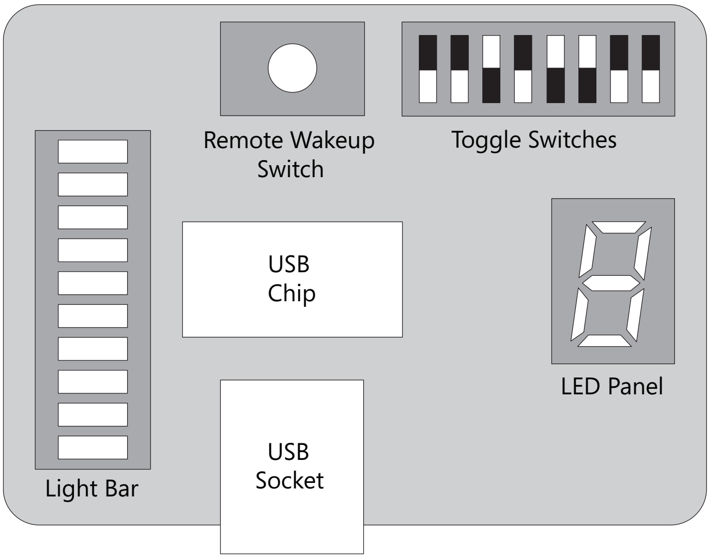

Figure 1-1 Simplified drawing of the USB learning device

The OSR Learning Kits include all of the required hardware specifications to implement a driver, plus sample code and test applications. See the OSR Web site for more information and to obtain kits–online at http://www.osronline.com.

Key Information Sources

In addition to the WDK samples and documentation, you can find numerous additional resources on the Microsoft Web site and through other Web-based community discussion sites.

WHDC—Windows Hardware Developer Central The WHDC Web site hosts a diverse collection of resources for driver developers and hardware manufacturers including white papers, tutorials, tips, and samples.

WHDC also hosts various publications that do not fit into the WDK or MSDN framework, including blogs by experts on the development team that address driver development issues.

You can subscribe to the Microsoft Hardware Newsletter to receive notice of new papers and kits, along with other information of interest to driver developers. See the WHDC Web site–online at http://www.microsoft.com/whdc.

Blogs Several members of the WDF development team maintain online blogs and write about various aspects of driver development. The WDF bloggers often respond to issues that developers are discussing on newsgroups and list servers. To view a list of blogs by Microsoft driver development experts, go to http://go.microsoft.com/fwlink/?LinkId=79579.

Microsoft Newsgroups for Driver Developers If you cannot find an answer in the WDK documentation, you can ask questions and share information through one of the MSDN newsgroups. Newgroups often provide information on issues that cannot be readily answered otherwise. The following newsgroups are dedicated to driver development:

- Microsoft.public.development.device.drivers
- Microsoft.public.windowsxp.device_driver.dev
- Microsoft.public.win32.programmer.kernel

To participate in these groups, you can access the newsgroups with Internet Explorer or another browser. Also, you can connect a newsgroup reader such as Outlook Express to the newsgroup server. The newsgroup server is at msnews.microsoft.com.

See the Microsoft newsgroup home page at http://msdn.microsoft.com/newsgroups/. The groups dedicated to driver developers are located under Windows Development \Windows DDK.

As part of the beta programs for WDF, Microsoft also moderates several news groups for KMDF, UMDF, and tools.

See "Hardware and Driver Developer Community" on the WHDC Web site for information about participating in WDF newsgroups and for information about WDF beta programs–online at http://go.microsoft.com/fwlink/?LinkId=79580.

OSR Online Open Systems Resources hosts three important list servers on the OSR Online Web site:

- **NTDEV— Windows System Software Developers List** Dedicated to the development of drivers for the Windows family of operating systems.
- **NTFSD—Windows File System Developers List** Dedicated to topics related to developing file systems or file system filter drivers.
- **WINDBG—Windows Debugger Users List** Dedicated to issues and changes related to use of WinDbg and debugger updates.

See the OSR Online Web site for information about how to subscribe–online at http://www.osronline.com/.

Channel 9 The Channel 9 Web site sponsored by MSDN contains a wide variety of information for developers including videos, podcasts, wikis, and forums. Although much of the site is devoted to application developers, the site provides some content that addresses driver development, Windows internals, and hardware issues—online at http://channel9.msdn.com/.

Conferences Microsoft hosts conferences that are dedicated to the interests of driver developers and hardware manufacturers, and several industry organizations and private companies provide training for driver developers. See the list at "Hardware and Driver Developer Community" on the WHDC Web site—online at http://go.microsoft.com/fwlink/?LinkId=79580.

WinHEC and Other Microsoft-Sponsored Events WinHEC is the Microsoft premier event for hardware engineers and designers, driver developers and testers, and business decision-makers and product planners. The conference is typically held every year in the spring.

See the WinHEC Web site at http://www.microsoft.com/whdc/winhec/.

Classes and Seminars Microsoft and several other companies present a variety of classes and seminars on topics that are of interest to driver developers.

See "Conferences and Training for Developers" on the WHDC Web site—online at http://go.microsoft.com/fwlink/?LinkId=79334.

Key References

WDF on WHDC	http://go.microsoft.com/fwlink/?LinkId=79335
WDK on WHDC	http://go.microsoft.com/fwlink/?LinkId=79337
Driver developer blogs	http://go.microsoft.com/fwlink/?LinkId=79579
Newsgroups	http://go.microsoft.com/fwlink/?LinkId=79580

Staying Up to Date with WDF

We continuously publish new information about WDF and the frameworks on the WHDC Web site. Check the "Developing Drivers with WDF: News and Updates" page on WHDC regularly for new code samples, white papers, news about the frameworks and the WDK, and any updates to this book. You'll find links to blogs for Windows driver developers. We've also posted hotlinks for all the references in this book, so you can quickly display cross-references without typing—online at http://go.microsoft.com/fwlink/?LinkId=80911.

—The WHDC Web Team, Microsoft

Chapter 3
WDF Fundamentals

The WDF driver model defines an object-oriented, event-driven environment for both kernel-mode (KMDF) and user-mode (UMDF) drivers. Driver code manages device-specific features, and a Microsoft-supplied framework calls the WDF driver to respond to events that affect the operation of its device.

This chapter introduces fundamental concepts for the design and implementation of WDF for UMDF and KMDF drivers.

In this chapter:

WDF and WDM 51
What Is WDF? 52
WDF Object Model 53
I/O Model 55
Plug and Play and Power Management 59
Security 61
WDF Verification, Tracing, and Debugging Support 63
Servicability and Versioning 63

For this chapter, you need ...	From ...
WDK documentation	
User-Mode Driver Framework Design Guide	http://go.microsoft.com/fwlink/?LinkId=79341
Kernel-Mode Driver Framework Design Guide	http://go.microsoft.com/fwlink/?LinkId=79342
Other	
Writing Secure Code, Howard and LeBlanc	http://go.microsoft.com/fwlink/?LinkId=80091

WDF and WDM

WDF drivers serve the same purpose as WDM drivers: they handle communication between Windows and a device. Although WDF represents an entirely new driver model, it is not distinct from WDM. WDF functions as an abstraction layer between WDM and the WDF driver that simplifies the task of implementing robust, secure, and efficient drivers.

WDF provides a framework that handles the key tasks of a WDM driver: it receives and handles IRPs, manages Plug and Play and power state changes, and so on. The framework calls on the client WDF driver to provide device-specific functionality. Although WDF supports two frameworks—UMDF and KMDF—the high-level design and functionality of both frameworks are quite similar.

This chapter provides a conceptual overview of WDF and WDF drivers, focusing on the basic features that both WDF frameworks have in common.

Chapter 4, "Overview of the Driver Frameworks," discusses how the two frameworks are implemented and the types of devices that they support.

What Is WDF?

WDF provides a unified driver model for a large range of device types. The most important features of the model include:

- Support for both user-mode and kernel-mode drivers.

  The core of the model can be used by either type of driver.
- A well-designed object model.

  WDF drivers interact with objects through a robust and consistent programming interface.
- An object hierarchy that simplifies object lifetime management and synchronization of I/O requests.
- An I/O model in which the frameworks handle interactions with the operating system.

  The frameworks also manage the flow of I/O requests, including responding to Plug and Play and power events.
- A Plug and Play and power management implementation that provides robust state management and intelligent default processing for state transitions.

  WDF drivers handle only those state transitions that are relevant to their device.

The framework provides a set of objects that represent various fundamental Windows and driver-related constructs, such as device, driver, I/O request, queue, and so on. WDF drivers use the framework objects to implement the various aspects of driver functionality. For example, when a WDF driver requires a queue to manage I/O requests, it creates a framework queue object.

Each object exposes a programming interface that the WDF driver uses to access object properties or direct the object to perform tasks. Each object also supports one or more events, which are raised in response to occurrences such as a system state change or the arrival of an I/O request. Events allow the framework to notify a WDF driver that something interesting has occurred and pass control to the WDF driver to handle the event. For example, a framework queue object raises an event to notify the WDF driver that an I/O request is ready to be processed.

Some framework objects are created by the framework and passed to the WDF driver. Other framework objects are created as the WDF driver needs them. As the developer, your job is to assemble the appropriate objects in a structure that supports the requirements of the device. The structure controls how the framework routes I/O through the WDF driver and how that driver interacts with the operating system and the device. The details of the structure vary, depending on the particular requirements of the device, but all WDF drivers are constructed from the same basic pieces.

Some framework objects are permanent or very long lived. For example, when a device is plugged in, a typical WDF driver creates a device object to represent the device and one or more queue objects to manage I/O requests. These objects usually persist for the lifetime of the device. Some objects are transient; they serve a specific short-term purpose and are destroyed as soon as that purpose is achieved. For example, a WDF driver could create an I/O request object to issue a single I/O request and then destroy it after the request is completed.

The framework provides default processing for all events, so WDF drivers are not required to handle any of them. However, depending solely on the framework's default event processing leads to a functional, but relatively uninteresting driver. To provide the necessary device-specific support, WDF drivers must handle at least some events.

To override the framework's default event handling, a WDF driver must explicitly register a callback. The framework invokes the callback to notify the WDF driver that the event has occurred and allow the driver to do any necessary processing. This opt-in event model is a key aspect of WDF. The framework provides intelligent default processing for all events—if the default processing for an event is adequate, WDF drivers do not need to register a callback. WDF drivers register callbacks for only those events for which the framework's default processing is not sufficient. All other events are handled by the framework, without any driver intervention.

WDF Object Model

The WDF object model defines a set of objects that represent common driver constructs such as devices, memory, queues, I/O requests, and the driver itself. Framework objects have well-defined life cycles and contracts, and a WDF driver interacts with them through well-defined interfaces. UMDF objects are implemented as COM objects, whereas KMDF objects are implemented as a combination of opaque "handles" and functions that operate on those handles. However, at a conceptual level, the WDF object models for KMDF and UMDF are similar.

Framework objects are created by both the framework and the client WDF driver. Whether the framework or the driver creates the object depends on the particular object and its intended use:

- Objects such as the file object are created by the framework and passed to the WDF driver.
- Objects such as the device object must be created by the WDF driver.
- Objects such as I/O request or memory objects can be created by either the framework or the WDF driver, depending on the circumstances.

Programming Interface

A framework object exposes a programming interface that consists of properties, methods, and events:

- **Properties** Properties represent the characteristics of an object. Each property has an associated method that gets the property value and, if relevant, a separate method that sets the value.
- **Methods** Methods perform actions on the object itself or direct the object to perform actions.
- **Events** Events are object-related occurrences that the WDF driver can choose to respond to, such as the arrival of an I/O request or a change in power state.

To handle an event, a WDF driver implements a callback and registers it with the framework:

- A UMDF driver implements COM-based callback objects, which expose callback interfaces to handle the events.
- A KMDF driver implements callback functions.

Regardless of how they are implemented, both types of callbacks work in much the same way. Because the framework implements default handlers for all events, WDF drivers register callbacks only for those events that are relevant to their device. When an event occurs:

- If a WDF driver has registered a callback for an event, the framework invokes the callback and the driver handles the event.
- If a WDF driver has not registered a callback for an event, the framework invokes the default event handler and applies default processing.

Object Hierarchy

Framework objects are organized in a hierarchy, which the frameworks use to manage issues such as object lifetime, object cleanup, and synchronization scope. For example, if an object has descendants, all descendants are deleted when the object is deleted. The hierarchy is not based on inheritance relationships between the various framework objects—it is based on the scope of the various objects and the order in which they must be destroyed. The hierarchy is defined as follows:

- The framework driver object is the root of the hierarchy; all other objects are its descendants.
- Some objects, such as queue objects, must always be children of the device object or of an object that is a descendant of the device object.
- Some objects, such as memory objects, can have one of several parents.

Chapter 5, "WDF Object Model," discusses the WDF object model in detail.

Concurrency and Synchronization

Managing concurrent operations is an issue for most programs. WDF simplifies the issue by implementing several internal synchronization mechanisms. In particular, a driver can direct the framework to hold a lock when the framework invokes a callback. The WDF object hierarchy supports a feature called synchronization scope—also called the locking constraint—that allows the WDF driver to specify which object's lock should be acquired when the framework calls the driver's I/O event callbacks.

Chapter 10, "Synchronization," discusses synchronization issues in general and synchronization scope in particular.

> **Tip** If you're writing a KMDF driver, you should also be familiar with IRQLs and kernel-mode synchronization and locking mechanisms. Chapter 15, "Scheduling, Thread Context, and IRQL," discusses these issues.

I/O Model

When Windows sends an I/O request to a WDF driver, the framework receives the request and handles the mechanics of dispatching, queuing, completing, and canceling requests on behalf of its drivers. When an I/O request arrives, the framework determines whether it should handle the request itself or invoke a callback to let the WDF driver handle the request. If the WDF driver is to handle the request, the framework packages the data into a framework request object and passes the object to the driver.

The framework keeps track of every I/O request. Because the framework is aware of all active requests, it can call the appropriate callbacks when an I/O request is canceled, the system's power state changes, the device is removed, and so forth.

A WDF driver manages the flow of I/O requests by creating one or more queue objects and configuring each object for:

- The type of I/O requests that the queue handles.
- How requests are dispatched from the queue.
- How power management events affect the queue.

WDF drivers register queue callbacks to receive requests, and the queue object dispatches requests by invoking the appropriate callback. A WDF driver can configure a queue object to dispatch requests in one of the following three ways:

- **Parallel** The queue object pushes requests to the driver as soon as they arrive. More than one request can be active at the same time.
- **Sequential** The queue object pushes requests to the driver, but does not dispatch a new request until the previous request has been completed or forwarded to another queue.
- **Manual** The driver pulls requests from the queue as needed.

Plug and Play and power management events can affect the state of I/O queues. The framework provides integrated Plug and Play and power management support for I/O request queues, and it integrates queuing with request cancellation. A WDF driver can configure a queue so that the framework starts, stops, or resumes queuing as appropriate in response to Plug and Play or power events. WDF drivers can also explicitly start, stop, resume, and purge queues, as required.

I/O Request Cancellation

Because Windows I/O is inherently asynchronous, canceled I/O requests are often difficult to handle correctly. A driver must cope with several potential race conditions that require one or more locks. WDM drivers must manage the necessary locks by themselves, and the required code is typically scattered among several driver routines. The framework provides default handling for I/O request cancellation by managing the locks for the I/O queues and by canceling queued requests without requiring driver intervention. WDF drivers that use the WDF defaults typically require little if any cancellation code.

With WDF, when an I/O request is canceled:

- By default, the framework manages cancellation for requests that are in a queue.
- Requests that have been removed from a queue and dispatched to a WDF driver cannot be canceled unless the driver has specifically marked them as cancelable.
- WDF drivers can specify whether the framework can cancel requests that the driver is actively processing.

This feature allows a WDF driver to easily support cancellation of long-running requests. The framework helps the WDF driver manage the inherent race conditions, and the driver is primarily responsible for the required code to cancel the request.

Chapter 8, "I/O Flow and Dispatching," discusses I/O.

I/O Targets

WDF drivers must sometimes send I/O requests to other drivers. For example:

- WDF drivers can sometimes process all I/O requests themselves, but many drivers must pass at least some of the requests that they receive down the stack for further processing.
- WDF drivers must sometimes initiate I/O requests.

  For example, function drivers sometimes send device I/O control requests down the stack to get information from the bus driver.
- WDF drivers must sometimes send I/O requests to an entirely different device stack.

  For example, a driver might require information from another device stack before it can complete a request.

WDF drivers send requests to an I/O target, which is a framework object that represents the driver that is to receive the request. The default I/O target for a WDF driver is the next lower driver in the device stack. However, I/O targets can also represent another driver in the same stack or a driver in an entirely different stack. WDF drivers can send a request to an I/O target synchronously or asynchronously. They can also specify a time-out value for either type of request to limit how long the framework will wait before canceling the request.

I/O target objects support a programming interface that WDF drivers use for purposes such as tracking the state of the target, formatting requests in a target-specific way, obtaining information about the target, and receiving notification if the target is removed. I/O target objects also track queued and sent requests, and they can cancel outstanding requests if changes occur in the state of the target device or the WDF driver that sent the request.

Chapter 9, "I/O Targets," discusses I/O target objects.

How to Handle Nonfatal Errors

WDF drivers call WDF methods for many different purposes. Many of these function calls can fail:

- Sometimes the error is so serious that the driver cannot recover.

  Such fatal errors cause UMDF to crash the host process and cause KMDF to issue a bug check.

- Some errors are less serious and do not affect device or driver operation to the extent that the driver cannot recover.

  In that case, the function reports the nature of the error to the driver by returning an appropriate status value—an HRESULT value for UMDF drivers or an NTSTATUS value for KMDF drivers. The driver can then handle the error as appropriate.

  Chapter 22, "How to Debug WDF Drivers," discusses fatal errors.

You must be scrupulous about checking return values for errors to ensure that they are handled properly. However, only WDF functions that return a status value can fail. All other functions are guaranteed to simply return a value of the appropriate type, although that return value could be NULL in some cases.

Sometimes the WDF driver itself detects errors. However, only those callbacks that return a status value must be concerned with returning errors. In that case, the callback reports nonfatal errors to the framework by returning the appropriate status value.

> **Tip** The PRE*fast* static analysis tool flags any instances where you failed to check a return status. Chapter 23, "PREfast for Drivers," provides details.

Reporting UMDF Errors

The HRESULT type supports multiple success and failure codes:

UMDF

- UMDF drivers can check the HRESULT values returned by UMDF methods for an error condition by passing the returned value to the SUCCEEDED or FAILED macro.

  If appropriate, the driver can examine the actual error code to determine how to proceed.

- UMDF driver callbacks that encounter an error should return the appropriate HRESULT value.

  The WDK reference page for each method lists the HRESULT values that the framework is prepared to receive.

Chapter 18, "An Introduction to COM," discusses HRESULT values in detail.

Reporting KMDF Errors

The NTSTATUS type also supports multiple success and failure codes:

- KMDF drivers can check the NTSTATUS values returned by KMDF functions by passing the returned value to the NT_SUCCESS macro.

  If appropriate, examine the actual error code to determine how to proceed.

- KMDF driver callbacks that encounter an error should return the appropriate NTSTATUS value.

  The WDK reference page for each function lists the NTSTATUS values that the framework is prepared to receive.

Note The Windows-defined NTSTATUS values are in Ntstatus.h, which is included with the WDK. It is also possible to define custom NTSTATUS values to handle scenarios that are not covered in Ntstatus.h. In general, custom values are used only in circumstances where both components can be expected to understand the value. Several custom NTSTATUS values are used to communicate status between a KMDF driver and the framework. If you complete an I/O request with a custom KMDF NTSTATUS value, KMDF remaps the custom error code to a well-known NTSTATUS value, which is then returned to the originator of the request.

The KMDF custom NTSTATUS values are defined in %wdk%\inc\wdf\kmdf\ *VersionNumber*\Wdfstatus.h.

Plug and Play and Power Management

Seamless and robust handling of Plug and Play and power events is critically important to system reliability and a good user experience. A primary WDF design goal was to simplify the implementation of Plug and Play and power management support and make this support available to both UMDF and KMDF drivers.

WDF drivers are not required to implement the complicated logic that is required to track Plug and Play and power state. Internally, the framework supports a set of state machines that manage Plug and Play and power state for WDF drivers. The framework notifies WDF drivers of changes in Plug and Play and power state through a series of events, each of which maps directly to device-specific actions that the driver can perform.

WDF Design Criteria for Plug and Play and Power Management

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 Plug and Play or power management event.

  Instead, the driver should be able to "opt in" and handle only those events that are relevant to its device, with the framework handling the rest.

- WDF actions at each point must be well defined and predictable.

  In effect, a contract applies to each Plug and Play and power event that clearly defines the driver's responsibilities.

- Plug and Play and power management events should be based on a group of core state changes, power-up, and power-down.

  Actions should be added to that core to handle specific scenarios. For example, a Plug and Play stop action invokes the core power-down event plus a release-hardware event.

- Plug and Play and power management events should be related to a well-defined task.

  Drivers should not be required to track system state to determine how to respond to a particular event.

- The frameworks should provide default behavior for a rich set of Plug and Play and power features, including resource rebalancing, device stop, device removal, device ejection, fast resume, shutdown of idle devices, and device wake-up by external signals.
- A driver should be able to override any framework-supplied defaults.
- Plug and Play and power management should be thoroughly integrated with other parts of the framework, such as controlling the flow of I/O requests.
- The frameworks must support both simple and complex hardware and driver designs.

  Simple tasks should be easy to implement, but the framework should provide straightforward ways for developers to extend their design to handle complex tasks.

- The frameworks should be extensible and flexible at the appropriate points, so that drivers can respond to state changes with their own custom actions.

  For example, WDF supports self-managed I/O, which can be used to coordinate operations that are not related to queued I/O requests or are not subject to power management for such state changes.

The framework provides default processing for every Plug and Play and power event. This approach vastly reduces the number of decisions that a WDF driver is required to make—especially during power transitions. WDF drivers contain much less Plug and Play and power management code than WDM drivers. Some WDF software drivers do not require any Plug and Play and power management code at all.

Chapter 7, "Plug and Play and Power Management," provides details about how WDF handles Plug and Play and power state changes.

Security

Every driver must be secure. Users trust drivers to transfer data between their applications and their devices. An insecure driver can expose sensitive data such as the user's passwords or account numbers to theft. Insecure drivers can also expose users to other security exploits such as denial of service or spoofing.

Drivers pose more security risks than applications, for the following reasons:

- Drivers can typically be accessed by any user and can be used by multiple users at the same time.
- Users are typically unaware that they are using a driver.

Security is closely related to reliability. Although the two requirements sometimes involve different programming issues, an insecure driver cannot be reliable, and vice versa. For example, a driver that crashes the system in effect causes a denial-of-service attack.

This book cannot cover every aspect of driver security; that would be a book in itself. However, understanding a few basics can make a big difference. The core precept of WDF driver security is that every driver must work to prevent unauthorized access to devices, files, and data. WDF is designed to contribute to driver security by providing safe defaults and extensive parameter validation.

This section briefly discusses WDF security features. For more information, see the WDK, which provides fundamental guidelines that apply to both UMDF and KMDF drivers.

See "Creating Reliable and Secure Drivers" in the WDK for security guidelines—online at http://go.microsoft.com/fwlink/?LinkId=80063. See also *Writing Secure Code, Second Edition,* by Howard and LeBlanc, for a general discussion of secure coding practices—available at http://go.microsoft.com/fwlink/?LinkId=80091.

Safe Defaults

Unless a WDF driver specifies otherwise, the framework requires administrator privileges for access to sensitive data such as exposed names, device IDs, or WMI management interfaces. The WDF default security settings provide secure names for kernel-mode device objects so that only kernel-mode components and administrators can access these objects.

Chapter 6, "Driver Structure and Initialization," provides information about device object security.

The framework automatically provides default handling for all I/O requests that the driver does not handle. The default handling prevents unexpected requests from entering a WDF driver that might misinterpret them. However, WDF does not protect drivers from ill-formed requests—when WDF drivers handle requests, they are responsible for handling them appropriately.

By default, UMDF drivers run in the LocalService account, which supports minimum security privileges. However, when a client's application opens a UMDF driver, it can grant the driver permission to impersonate the client. That action allows the driver to perform tasks that require a higher level of privilege.

Chapter 8, "I/O Flow and Dispatching," discusses impersonation.

Parameter Validation

Buffer overflows are a major source of security problems in drivers as well as in applications. All WDF functions or methods that use buffers have a length parameter that specifies the required minimum buffer length. For I/O requests, WDF extends this practice by placing I/O buffers in framework memory objects. The data access methods exposed by memory objects automatically validate buffer length and determine whether the permissions that are specified for the buffer allow write access to the underlying memory.

One of the most common driver security problems occurs when kernel-mode drivers fail to properly handle user-mode buffers that I/O control requests carry. This is particularly true for requests that specify neither-buffered-nor-direct I/O (that is, METHOD_NEITHER I/O). Because the pointers to user-mode buffers in this type of request are inherently unsafe, KMDF drivers do not have direct access to the pointers by default. Instead, WDF provides methods to allow a WDF driver to safely probe the buffer and lock it into memory.

WDF Verification, Tracing, and Debugging Support

WDF includes several testing and tracing features to help you to find problems early in the development cycle, including the following:

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

WDF includes verifiers for both frameworks to support framework-specific features that are not currently available in Driver Verifier (Verifier.exe). The WDF verifiers issue extensive tracing messages that supply detailed information about activities within the framework itself. They also track references to each WDF object and build a trace log that can be viewed by using WinDbg.

Chapter 11, “Driver Tracing and Diagnosability,” explores tracing.

WDF supports two sets of debugger extensions for the WinDbg debugger: one for debugging UMDF drivers and one for debugging KMDF drivers. The extensions supplement the basic capabilities of WinDbg by providing detailed information on WDF-specific issues.

Chapter 22, “How to Debug WDF Drivers,” discusses debugging.

Serviceability and Versioning

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 nonstandard way—is likely to encounter compatibility problems from one Windows release to the next.

WDF drivers are less susceptible to such problems because Microsoft tests the frameworks on each new version of the operating system. This testing ensures that WDF drivers maintain consistent behavior from one release to the next.

To improve driver serviceability and help to prevent compatibility problems, WDF provides versioning and support for side-by-side operation of different framework major versions. Each release of the framework has a major and minor version number. A WDF driver always uses the major version for which it was built and tested. If a newer major version of the framework is installed, the older version runs side by side with the newer major version and WDF drivers continue to bind to the major version for which they were built.

WDF drivers specify the major and minor version of the framework that they were built for when they are compiled. When the driver is installed, it specifies the version of the framework that must be present on the system to meet the driver's minimum requirements:

- If the specified major version is not present on the user's system, the WDF co-installer that is included in all WDF installation packages automatically installs the specified major version side by side with any older major versions.
- If the specified minor version is more recent than the one on the user's system, the WDF co-installer updates the framework to the newer minor version.

  The older minor version is backed up, so that users can use System Restore to return to it, if necessary.

When a new minor version is installed on the system, the existing WDF drivers that use the same major version automatically bind to the new minor version. This means that a WDF driver binds to the most recent minor version of the major version against which the driver was compiled. This feature allows WDF drivers to benefit from bug fixes in minor versions.

> **Tip** Even drivers that comply with WDF rules could be affected by subtle changes between versions of Windows. You should always test your drivers with service pack candidates and new versions of Windows to ensure that there are no problems. Chapter 20, "How to Install WDF Drivers," discusses WDF versioning and installation.

Why Did I Choose to Work on WDF?

During the development of Windows 2000 (circa 1998) when I was learning how to write a driver and debug kernel-mode code, I was constantly amazed that the whole system worked. I saw three different levels of complexity in the driver: first, with communicating with its hardware; second, with implementing Plug and Play and power and other operating system requirements; and third, with communicating with other drivers.

It made my head spin, thinking of how hard it was just to get one of these right, let alone implementing each of them and then managing the interaction between these three areas. Even back then, I knew there had to be a better way to develop a driver, but I didn't have the experience or the time to do something about it. So when I heard about the driver frameworks team, I knew I had to work on the team to fix these very issues and make driver writing a simpler task.

–Doron Holan, Windows Driver Foundation Team, Microsoft

Part 2
Exploring the Frameworks

In this part:

Chapter 4: Overview of the Driver Frameworks 67
Chapter 5: WDF Object Model 91
Chapter 6: Driver Structure and Initialization 129

Glossary

access right A permission granted to a process to manipulate a specified object in a particular way by calling a system service. Different system object types support different access rights, which are stored in an object's access control list.

Active Template Library (ATL) A set of template-based C++ classes that are often used to simplify the creation of COM objects.

alternate setting In a USB interface, a collection of endpoints that describe a function for the device.

annotation A macro that can be applied to source code to help PRE*f*ast analyze the code more effectively.

arbitrary thread context The context of the thread that happens to be running on a processor when the system borrows the thread to run a driver routine, such as an ISR. The driver routine can make no assumptions about the contents of the address space.

asynchronous I/O A model for I/O in which the operations carried out to satisfy I/O requests do not necessarily occur in sequence. The application that originally made the request can continue executing rather than waiting for its I/O to complete, the I/O manager or a higher-level driver can reorder I/O requests as they are received, and a lowest-level driver can start an I/O operation on a device before it has completed the preceding request, particularly in a multiprocessor machine.

ATL Active Template Library

atomic operation An operation that must run to completion without interruption.

boot-start driver A driver that is installed during the boot procedure and is required to start the system.

bug check An error that is generated when core Windows data structures have been irretrievably corrupted, sometimes referred to as a system crash.

bus driver A driver that enumerates the devices that are attached to bus.

.cab file cabinet file

cabinet file A "cabinet" of compressed installation files, with a file extension of .cab.

callback object A driver-created object on which a UMDF driver implements the callback interfaces that are required to service events raised by one or more framework objects.

checked build A build that has been compiled with debug symbols and built with special support for debugging. Checked builds are used only for testing and debugging.

class driver A driver that typically provides hardware-independent support for a class of physical devices and is supplied by Microsoft.

class factory A specialized COM object that clients use to create an instance of a particular COM object.

class GUID See *setup class GUID*.

class ID (CLSID) A GUID that uniquely identifies a particular COM object. CLSIDs are required for COM objects that are created by a class factory, but are optional for objects that are created in other ways.

CLSID class ID

co-installer A DLL that augments the device installation operations performed by a class installer.

collection object A KMDF object that maintains a linked list of other KMDF objects of any types.

COM Component Object Model, which is a platform-independent, distributed, object-oriented system for creating binary software components that can interact.

COM client A process that uses COM objects.

COM server A COM object that provides services to clients.

common-buffer DMA A DMA design in which the driver allocates a buffer in system memory that it shares with the DMA device. Sometimes referred to as "continuous DMA."

composite annotation An annotation composed of two or more primitive annotations and other composite annotations.

concurrency The simultaneous execution of two code sequences.

context area A driver-defined area within a WDF object in which the driver stores information that is specific to that instance of the object.

control device object (1) A device object that is not part of the Plug and Play device stack. (2) When capitalized, a device object that is the target for I/O requests between the reflector and the UMDF driver manager.

critical region A locking mechanism that prevents the delivery of most asynchronous procedure calls. A thread running in a critical region is running at an intermediate IRQL level between PASSIVE_LEVEL and APC_LEVEL. Sometimes called a "critical section."

critical section See *critical region*.

current priority A thread's priority at any given time.

deadlock A runtime error condition that occurs when two threads of execution are unable to continue running because each is waiting to acquire a resource that the other holds.

DDI device driver interface

debugger command A WinDbg command-line utility that performs various basic debugging operations.

debugger extension A WinDbg command-line utility that provides functionality beyond that provided by debugger commands. They are useful in debugging special-purpose software such as WDF drivers.

deferred procedure call (DPC) A queued call to a kernel-mode function that runs at DISPATCH_LEVEL.

device bus logical address space The address space for a device bus.

device driver interface (DDI) A collection of system-supplied routines that a driver calls to interact with system services. DDI is the driver equivalent of API.

device ID A vendor-defined device identification string that is the most specific ID that Setup uses to match a device to an INF file.

device I/O control request An I/O request that cannot be represented as a read or write request and typically involves some other hardware operation.

device instance An individual unit of hardware. For example, if Company ABC manufactures a CD-ROM drive with a model name of XYZ, and if a particular system includes two of these drives, then there are two instances of device model XYZ.

device instance ID A system-supplied device identification string that uniquely identifies an instance of a device in the system.

device interface Device functionality that a driver exposes to applications or other system components. Each device interface is a member of a system-defined or vendor-defined device interface class.

device interface class A group of interfaces that generally apply to a particular type of device and are the means by which drivers make devices available to applications and other drivers.

device interrupt request level (DIRQL) The range of IRQLs associated with device interrupts. The exact range of DIRQLs depends on the processor architecture.

device node See *devnode*.

device object An object that represents a driver's participation in the processing of I/O requests for a particular device.

device power policy owner The driver that controls the power policy for a device, determining when it sleeps and when it wakes.

device power state The level of power consumption by a device. Device power states range from D0 to D3, where D0 is the fully-powered working state and D3 is the powered-down (off) state.

device setup class A group of devices that are set up and configured in the same way.

device stack A collection of device objects and associated drivers that handle communication with a particular device.

device synchronization scope A WDF synchronization mechanism in which the framework acquires a presentation lock at the device object level.

devnode An element of the PnP manager's device tree. The PnP manager uses a device stack's devnode to store configuration information and track the device.

direct memory access (DMA) A method of transferring data between a device and main memory without intervention by the CPU.

DIRQL device interrupt request level

dispatcher (1) The system's thread scheduler. (2) A UMDF component that directs I/O requests to I/O targets that are not part of the UMDF device stack.

dispatch execution level (DISPATCH_LEVEL) The IRQL at which the Windows thread dispatching code, paging code, and many kernel-mode driver functions run.

DMA direct memory access

DMA transaction A complete I/O operation that involves DMA, such as a single read or write request from an application.

DMA transfer A single hardware operation that transfers data from main memory to a device or from the device to main memory.

Down device object The reflector's device object that receives requests from the UMDF driver host process and passes them to the kernel-mode device stack.

DPC deferred procedure call

driver manager The UMDF component that creates and shuts down the driver host processes, maintains status information about them, and responds to messages from the reflector.

driver object An object that represents a driver.

driver package An installation package that includes the driver binary and all of its supporting files.

driver stack A chain of drivers that is associated with one or more device stacks to support the operations of devices.

dynamic analysis The examination of a program while executing its code. Unit tests and stress tests are examples of dynamic analysis.

dynamic verification The use of dynamic analysis to verify that the program complies with a specification or protocol.

elevated privileges In Windows Vista, the level of privileges that is required to perform certain operations.

endpoint A target for input, output, or control operations on a USB device.

event A notification that something has happened that a driver might need to respond to, such as the arrival of an I/O request.

exclusive device A device for which only one handle can be open at a time.

execution level For KMDF drivers, an attribute of some objects that limits the IRQL at which the framework invokes certain object callback functions.

exception A synchronous error condition that results from the execution of a particular machine instruction.

fatal error An error from which the driver or system cannot recover.

FDO functional device object

file object A WDF object that represents a single use of an individual file and maintains state information for that use.

filter [in PRE*fast* log] An option in the PRE*fast* defect log viewer that shows or hides particular messages.

filter device object (filter DO) A device object that represents a device for a filter driver.

filter driver A driver that modifies or monitors I/O requests as they pass through the device stack.

framework object An object managed by WDF.

free build A build used for released products. The free build is smaller in size and faster than a checked build, but is more difficult to debug.

function driver The primary driver for a device.

function role type A declaration that provides SDV with information about the intended use of a WDF callback function or WDM dispatch function.

function type class A category that PRE*fast* uses to perform stricter type matching for assignments of callback functions to function pointers and to apply checks that are specific to a particular function type.

functional device object (FDO) A device object that represents a device for a function driver.

guard An implicit condition imposed on a pointer argument of an action procedure for analysis with SDV.

GUID globally unique identifier

handle An opaque type through which a KMDF driver identifies a KMDF object.

hardware ID A vendor-defined device identification string that Setup uses to match a device to an INF file.

hibernate state System power state S4, in which power is off, but the system can resume and quickly restore its state from a file written to disk before power-down.

HID A human interface device, such as a keyboard or mouse.

host process The process, Wudfhost, in which a UMDF driver runs, along with related UMDF user-mode components such as the framework.

I/O completion routine A driver routine that runs after an I/O request has been completed by a lower driver in the stack.

I/O control code (IOCTL) A control code that is used to identify a particular device I/O control operation.

I/O request packet (IRP) A data structure used to pass a packet of data and related information between the I/O manager and the components of a device stack.

I/O target A WDF object that represents a target for an I/O request.

IDL interface definition language

IID interface identifier

impersonation The ability of a user-mode process to run with the security credentials of a particular user.

INF A text file that contains data used to install a driver.

in-flight request A request that has been dispatched to a driver but has not yet been completed or requeued.

in-process COM object (InProc object) A COM server that runs in the context of its client process.

instance ID A device identification string that distinguishes a device from other devices of the same type on a machine.

interface definition language (IDL) file A file that defines COM interfaces and objects in a structured interface definition language.

interface ID (IID) A GUID that uniquely identifies a particular COM interface. An interface always has the same IID, regardless of which object exposes it.

internal device I/O control request A type of device I/O control request that can be issued only by a kernel component.

interrupt A notification to the system that something—such as a hardware event—has occurred outside normal thread processing and must be handled as soon as possible.

interrupt object A KMDF object that represents the connection of a hardware interrupt source and a driver's interrupt service routine to the system's interrupt dispatch table.

interrupt request level (IRQL) A value that Windows assigns to each interrupt. In case of conflict, the interrupt with the higher IRQL has priority and the routine that handles it runs first.

interrupt service routine (ISR) A function implemented by a device driver to handle hardware interrupts.

interrupt spin lock A synchronization object that can be used at DIRQL.

INX file An architecture-independent INF file.

IOCTL I/O control code

IRP I/O request packet

IRQL interrupt request level

ISR interrupt service routine

kernel address space A block of virtual memory that is dedicated to the use of kernel-mode code.

kernel mode The operating mode in which the Windows core operating system and many drivers run.

kernel subsystems Components of the Windows kernel that support core features such as Plug and Play and expose the DDI routines that allow kernel-mode drivers to interact with the system.

KMDF Kernel-Mode Driver Framework

locking constraint See *synchronization scope*.

logical address space See *device bus logical address space*.

lookaside list A KMDF object that represents a driver-created, system-managed list of fixed-size buffers that can be allocated dynamically.

manual dispatch A queue dispatching technique in which a driver retrieves requests from a queue at its own pace by calling a method on the queue.

map register An internal structure used during DMA to alias a device-accessible logical page to a page of physical memory.

MDL memory descriptor list

memory descriptor list (MDL) An opaque structure, defined by the memory manager, that uses an array of physical page frame numbers to describe the pages that contain a virtual memory range.

MSI message The information that is passed with a message-signaled interrupt.

MSI message-signaled interrupt

mutex A PASSIVE_LEVEL synchronization object that provides mutually exclusive access to a shared region of memory.

neither I/O Neither buffered nor direct I/O.

NMI nonmaskable interrupt

nonmaskable interrupt (NMI) An interrupt that cannot be overruled by another service request. A hardware interrupt is called nonmaskable if it bypasses and takes priority over interrupt requests generated by software, the keyboard, and other devices.

nonpaged pool A region of kernel-space memory that is always physically resident and is never paged out to disk.

noise PRE*f*ast messages that might not represent actual errors in code, also referred to as "false positives."

non-power-managed queue A queue that is not managed with the respect to the Plug and Play and power state of the parent device object. Such a queue continues to dispatch requests after the device leaves the working state.

NTSTATUS A type used to return status information by many kernel-mode routines.

object manager A kernel subsystem that manages kernel objects.

object synchronization lock See *presentation lock*.

page fault An event that occurs when a process attempts to access data or code that is not resident in memory.

paged pool A region of kernel-space memory that is not physically resident and therefore can be written to the hard drive if necessary and then read back in when needed.

parallel dispatch A queue dispatching technique in which a queue delivers I/O requests to the driver as soon as possible, whether or not another request is already active in the driver.

passive execution level (PASSIVE_LEVEL) The operating level at which no interrupts are masked off. This is the level at which most applications and system software run.

PDO physical device object

physical device object (PDO) A device object that represents a device for its bus driver.

pipe An endpoint on a USB device that is configured in the current alternate setting in an interface.

Plug and Play (PnP) A combination of system software, device hardware, and device driver support through which a computer system can recognize and adapt to hardware configuration changes with little or no intervention by an end user.

PnP manager A kernel subsystem that manages Plug and Play operations.

pool tag A four-byte character literal that is associated with a dynamically allocated chunk of pool memory. A driver specifies the tag when it allocates the memory.

post-state The PRE*f*ast state of analysis just after a function call has returned.

power-managed queue A queue that the framework manages in accordance with the Plug and Play and power state of the parent device object. Such a queue stops dispatching requests when the device leaves the working state and resumes dispatching requests when the device reenters the working state.

power policy The set of rules that determine how and when a system or device changes power state.

power state The level of power consumption for the system or for an individual device.

pre-state The PRE*f*ast state of analysis just before a function call is made.

presentation lock A framework-created lock for device and queue objects that the framework uses to implement synchronization scope.

primitive annotation An annotation from which more complex annotations can be constructed.

priority An attribute of a thread that determines when and how often the thread is scheduled to run.

priority boost A set of system-defined constant values that are used to increase the thread priority of the requesting application when drivers complete an I/O request.

probe To check that a memory address range is in user-mode address space and that the range can be read or written in the current process context.

property store An area in the registry in which a UMDF driver can store information about its device.

queue synchronization scope A WDF synchronization mechanism in which the framework acquires a presentation lock at the queue object level.

race condition A situation in which two or more routines attempt to access the same data and the result of the operation depends upon the order in which the access occurs.

raise an exception A deliberate transfer of control to an exception handler when an exception occurs. A kernel-mode component, including any kernel-mode driver, cannot raise an exception while running at IRQL >= DISPATCH_LEVEL without crashing the system.

raised IRQL Any interrupt request level that is greater than PASSIVE_LEVEL.

raw resource list A list of hardware resources that the PnP manager assigns to a device. The raw resource list reflects the physical design of the device.

reflector A filter driver at the top of the kernel-mode device stack that facilitates communication between the kernel-mode device stack and all of the UMDF drivers on the system.

request completion callback A driver-implemented function that is called at the driver's request after the completion of an I/O request that the driver created or sent down the stack.

root-enumerated device A device whose devnode is a child of the root device in the PnP manager's device tree.

scatter/gather DMA A form of DMA in which data is transferred to and from noncontiguous ranges of physical memory.

scatter/gather list A list of one or more paired base addresses and lengths that describe the physical locations from which to transfer data in scatter/gather DMA.

SDV Static Driver Verifier

security descriptor A data structure used to hold per-object security information, including the object's owner, group, protection attributes, and audit information.

security violation An attempt by a user-mode process to access an object without having the correct access rights for the requested operation.

sequential dispatch A queue dispatching technique in which a queue delivers I/O requests to the driver one at a time, only after the previously dispatched request has been completed or requeued.

serialization The management of two concurrent items of the same class, such as callback functions or I/O requests, to prevent their concurrent operation.

setup class GUID A GUID that identifies a device setup class.

sleep state A system or device power state other than the system or device working state, S0 or D0.

software driver A driver that does not control any hardware, either directly or through a protocol such as USB.

spin lock A synchronization object that ensures mutually exclusive access at DISPATCH_LEVEL.

static analysis A testing method that involves examining the source or object code of a program without executing the code, typically to detect coding errors.

static verification The use of static analysis to verify that the program complies with a specification or protocol.

structured exception handling A system feature that supports controlled transfers to exception handlers when certain runtime exceptions occur.

surprise removal The unexpected removal of a device by a user without using the Device Manager or Safely Remove Programs application.

symbolic link (1) An instance of the symbolic link object type, representing a "soft alias" that equates one name to another within the object manager's name space. (2) A file object with special properties. When a symbolic link file is encountered as a component of a path name, rather than opening the file itself, the file system is redirected to a target file.

system working state System state S0, the fully on, fully operational power state.

symbol file A file that contains information about an executable image, including the names and addresses of functions and variables.

symbolic link name A name for a device that is typically created only by older drivers that run with applications that use MS-DOS device names to access a device.

synchronization A general term for the management of operations that share data or resources to ensure that access to such shared resources occurs in a controlled manner.

synchronization scope A configurable object-based mechanism through which a driver can specify the degree of concurrency for certain object callbacks.

SYS The file extension used for kernel-mode driver binaries.

system event log A file that contains a record of system events.

system power state The level of power consumption by the system as a whole. System power states range from S0 to S5, where S0 is the fully-on working state and S5 is the completely powered-down state.

thread context The operating environment that is defined by the values in a thread's CONTEXT structure, including the security rights and the contents of the address space.

thread interruption The mechanism by which Windows forces the current thread to temporarily run code at a higher interrupt level.

thread preemption The replacement by Windows of the running thread with another thread, usually of higher thread priority, on the same processor.

thread priority A scheduling priority for a thread, which has a value from 0 to 31, inclusive. Higher numbers indicate higher priority threads.

timer object A KMDF object with which a driver can request a callback at repeated intervals or only once after a specified time period has elapsed.

trace consumer An application that formats and displays trace messages.

trace controller An application that enables tracing and connects the trace provider with the trace consumer.

trace log A series of trace messages.

trace message A string generated by a trace provider that contains trace information.

trace provider A component that generates trace messages.

translated resource list A list that the PnP manager generates for each device, indicating where each of the device's hardware resources is mapped on the current system.

Up device object The reflector's target for I/O requests from the I/O manager.

UMDF User-Mode Driver Framework

USB interface A collection of alternate settings that describe variations on a function for a USB device. Only one alternate setting can be in use at any given time.

user mode A restricted operating mode in which applications and UMDF drivers run that does not permit direct access to core Windows routines or data structures.

virtual function table (VTable) An array of pointers to the implementation of each of the methods that are exposed by a COM object's interfaces.

wait lock A PASSIVE_LEVEL synchronization mechanism that KMDF implements through the WDFWAITLOCK object.

waking The transition from a sleeping state to the working state.

WDF Windows Driver Foundation

WDM Windows Driver Model

Windows Driver Model (WDM) The earlier Windows driver model, which supports the Windows NT family of operating systems, starting with Windows 2000.

work item (1) A mechanism—typically used by high-IRQL routines—to perform preferred processing at PASSIVE_LEVEL in a system thread. (2) A KMDF object that implements work item functionality.

x86 Refers to systems with 32-bit processors that run the Intel instruction set.

x64 Refers to systems with a processor architecture based on the x86 architecture with extensions for the execution of 64-bit software, including AMD64 processors and Intel processors with Extended Memory 64 Technology (EM64T).

Index

A

abstraction phase, SLAM engine, 830
access, synchronized. *See* synchronization
access rights, I/O target objects, 315
AcquireLock method (IWDFObject), 386
Active Template Library (ATL), 595
adaptive time-outs (UMDF), 297
/adddriver parameter (Driver Verifier), 681
AddRef method (IUnknown), 113, 589, 590–591, 600, 609–610
addressing capability (DMA devices), 554, 555
aliasing memory, annotations for, 791–792
alignment requirement (device property), 145, 554
/all parameter (Driver Verifier), 680
allocating memory, 44, 442–447. *See also* memory
  annotations for, 790–791, 798–799
  tagged memory allocation, 44
analysis mode (PRE*f*ast), 734
!analyze debugger extension, 715
annotation modifiers, 764–765
annotations in source code
  best practices, 816–818
  driver annotations, 774–778
  eliminating coding noise with, 751–752
  general-purpose, 760–774
  how to use, 752–760
  Osrusbfx2 sample driver (example), 818–822
  for SDV (Static Driver Verifier), 832–834
APC_LEVEL IRQLs, 511, 513, 514
APCs (asynchronous procedure calls), 514
Application Verifier, 693–694
applications
  I/O. *See* Windows I/O model
  in WDF architecture, 74
  in Windows system architecture, 26
AppVerif.exe, 693–694
arbitrary thread context, 39–40, 47, 510–511
ASSERT macro, 749
Assign functions, 96, 104
AssignContext method (IObjectCleanup), 124, 454
AssignDeviceInterfaceState method, 152
assumptions, making explicit, 749
asynchronous I/O, 27, 31–32, 339–340. *See also* synchronization
asynchronous procedure calls (APCs), 514
ATL (Active Template Library), 595
attributes, object, 94, 108
AutoForwardCreateCleanupClose method (IWDFDeviceInitialize), 150, 272, 274, 281
automatic forwarding of I/O requests, 272–273
automatic serialization, 395–396, 539
  timer objects, 460

B

BARs (base address registers), 534
base objects, 97, 454–455
  deleting, 116
  in object hierarchy, 112
base priority (threads), 508
Basic_SimpleIO.wsf script, 676
BATTERY scenario (PwrTest), 675
_bcount annotation, 761, 766–768
binary files, 623, 642, 661
binding to WDF framework, 639–640
blocking threads, 46
blogs for more information, 21
boosting thread priority, 509
boot-start drivers, signing, 654
Boot.ini file, 712
breaking into debugger, 688
BufAfterReqCompletedXxx rules, 851
buffer pointer lifetime, 252–253
buffer-size annotations, 765–771
buffered I/O, 34–35, 228, 229
  retrieving buffers in KMDF drivers, 249
buffers
  allocating for driver-created I/O requests, 323–330. *See also* allocating memory
  annotations for, 768–771
  DMA transfers, 552, 554, 560–562
  lifetimes of, 324–325
  lookaside lists, using, 445–446
  retrieving, 245–252
bug checks, 47, 692, 728
Build utility, 619–620
  flags for, 623
  supporting files, 620–622
building WDF drivers, 617–633
  build environments and supporting files, 619–622
  debugging. *See* debugging WDF drivers
  Fx2_Driver sample driver, 625–629
  general considerations, 618–619
  Osrusbfx2 sample driver, 630–633
  projects, 622–624
  testing best practices, 694–695
builds of Windows, obtaining, 17–18
bulk transfers (I/O model), 357–358
bus drivers, 29, 142, 144
  create requests, not accepting, 279–280
  enumeration of child devices, 157–158

bus drivers (*continued*)
  I/O request flow through, 237
  Plug and Play basics, 167–168
  power management, 169
bus-master devices, 550. *See also* DMA (direct memory access)
buses, levels of, 29–30

C

C_DEFINES macro, 627, 632
C++ programming for UMDF drivers, 87
callbacks, 54, 75, 93–94
  cleanup callbacks, 111, 117
  context data and, 123–124
  creating callback objects, 106–107
  destroy callbacks, 118
  I/O event callbacks, 271–291
    automatic forwarding, 272–273
    for create requests, 273–280, 280–281
    file object creation, 271–272
    I/O targets (KMDF), 320–321
    for read/write and IOCTL requests, 282–291
    self-managed, 298
  KMDF drivers, 72–73
  lifetime of callback objects, 113
  per role type (SDV), 837–838
  Plug and Play and power management, 171–173, 176–189. *See also* Plug and Play; power management
    serialization, 385
  requesting at intervals. *See* timer objects
  serialization, synchronization scope and, 387–398
    KMDF drivers, 391–398
    UMDF drivers, 390–391
  Skeleton sample driver, 481–486
  UMDF drivers, 70, 585–586, 611–613
    callback interfaces, 96–97
  when to implement, 76
  WMI instance events, 470–473
calling thread context, 510–511
cancel spin lock, annotations for, 795–796
canceled I/O requests, 56–57, 293–296, 351–352
  adaptive time-outs (UMDF), 297
  annotations for, 806
  request-cancellation rules (KMDF), 850
  synchronization of cancellation, 401–409
CancelIoEx function (Windows), 351
CancelSentRequest method (IWDFIoRequest), 351
CancelSentRequestsForFile method (IWDFIoTarget), 351
.cat files, 643, 653–654
categorizing trace messages, 419
CClassFactory class, 480
Channel 9 Web site, 22
checked builds of co-installers, 640
checked builds of drivers, 620, 700
checked builds of Windows, 47
  obtaining, 17–18
  testing drivers, 668
__checkReturn annotation, 761, 773–774
child devices
  child devices, 143, 157–159
  enumerating, 143, 157–159
  startup and, 180–183
child list objects, 102, 114
  creating, 157
  deleting, 115, 121
child objects. *See* hierarchy of WDF objects
ChkINF utility, 671
circular references, 113–114
class factories, 603–606
class IDs (CLSIDs), 592
class relationships (COM), 586
classes on driver development, 22
[ClassInstall32] section, INF files, 644
cleaning up tracing, 429–431
cleanup callbacks, 111, 117
cleanup of objects, defined, 112
cleanup requests (I/O), 224–225, 242
  automatic forwarding, 272–273
  event callbacks for, 280–281
  unbalanced with create requests, 273
Cleanup4CtlDeviceRegistered rule, 848
CLOCK_LEVEL IRQLs, 512
CLOCK2_LEVEL IRQLs, 512
close requests (I/O), 224–225, 242
  automatic forwarding, 272–273
  event callbacks for, 280–281
  unbalanced with create requests, 273
Closed state (I/O targets), 319, 340
CloseHandle function, 224–225
CLSIDs (class IDs), 592
CM_PARTIAL_RESOURCE_DESCRIPTOR structure, 532
CmResourceXxx types, 533, 536
co-installer packages, 14, 640–642
  incorrect version errors, 665
  INF files and, 646–648
coding errors, detecting. *See* PRE*f*ast for Drivers
coding noise, reducing, 749–752, 815
collection objects, 102, 114, 455–459
  deleting, 115
COM (Component Object Model), 583–613
  as basis for UMDF, 69, 480–481
  callback objects, implementing, 606–613
  DLL infrastructure, implementing, 478–480, 600–606
  overview of, 586–597
  using UMDF COM objects, 597–600
Command window (WinDbg), 700–701
command window, opening with elevated privileges, 15
common-buffer DMA devices, 551, 553, 566

compatibility problems. *See* serviceability
Complete method (IWDFIoRequest), 292
completed I/O requests, 121
CompleteWithInformation method (IWDFIoRequest), 292, 343
completing I/O requests, 291–293. *See also* completion routines (I/O)
  adaptive time-outs (UMDF), 297
  parent objects and, 121
  request completion information objects, 98
  request completion parameters objects, 98–99
  USB targets, 99
completion routines (I/O), 34, 47, 238–239. *See also* completing I/O requests
  DMA drivers, 575–577
  for I/O requests, 334–336
CompletionRoutine callback, 391
Comsup.cpp and Comsup.h files, 476
concurrency, 39–41
  WDF object hierarchy, 55
conditional driver annotations, 779–783
conditions on trace messages, 418–419
conferences for driver developers, 22
configuration descriptors for USB devices, 356–357
configuration structures for KMDF objects, 107
Configure public helper method, 485
ConfigureRequestDispatching method, 263–264
connecting interrupts, 540
constants as parameters, checking, 788
context areas
  for device objects (KMDF), 154
  type declaration (KMDF), 127
context, object, 95, 122–128
  KMDF drivers, 126–128
  UMDF drivers, 122–125
CONTEXT structure (threads), 509
context, thread, 509–511. *See also* threads
ContextSizeOverride attribute, 108
ContextTypeInfo attribute, 108
contiguous data buffers, 560
continuous reader (USB devices), 375–376
control codes, 226–227
control device objects, 79, 140, 143–144
  cleanup rules (KMDF), 848
control GUIDs, 425–427
control requests, 33, 48, 225–226, 242
  control codes for, 226–227, 228
  event callbacks for, 282–291
  formatting for I/O targets, 331–333
  internal, 228, 242, 322
    formatting for I/O targets, 332–333
    synchronous, 337
  synchronous, 337
ControlDeviceDeleted rule, 848
ControlDeviceInitAPI rule, 848
converting time values, 461
CopyFromBuffer method (IWDFMemory), 246, 444
CopyFromMemory method (IWDFMemory), 444
CopyToBuffer method (IWDFMemory), 246, 444
core Windows system architecture, 25–27
CPU architecture, selecting, 620
  INFs for, 645–646
crash, debugging, 710, 715–716
crash dump analysis, 698, 728
CrateWdfMemory method (IWDFDriver), 327
create requests (I/O), 224, 242
  automatic forwarding, 272–273
  event callbacks for, 273–280
  unbalanced, with cleanup/close, 273
CreateDevice method (IWDFDriver), 149, 151
CreateDeviceInterface method (IWDFDevice), 152
CreateFile function, 224
CreateFileHandleTarget method, 353–354
CreateInstance method (IClassFactory), 132, 135–137, 480–481, 603, 604–605
CreateInstance public helper function, 481–482, 484
CreateIoQueue method (IWDFDevice), 197, 198, 261
Create*Object* methods, 106
CreatePreallocatedWdfMemory method (IWDFDriver), 327, 329, 443–444
CreateRequest method (IWDFDevice), 322, 598–599
CreateUsbTargetDevice method (IWDFUsbTargetFactory), 359, 362
CreateWdfFile method (IWDFDevice), 312
CreateWdfMemory method (IWDFDriver), 443
CreateWdfObject method (IWDFDriver), 454
critical regions
  annotations for, 795–796
  PASSIVE_LEVEL IRQLs in, 514
.ctl files, generating, 434
CtlDeviceFinishInItDeviceAdd rule, 847
CtlDeviceFinishInItDrEntry rule, 847
CUnknown class, 480
current IRQL, finding, 522
current priority (threads), 508
custom installation applications, 658
custom trace message functions, 419. *See also* tracing

D

D3hot and D3cold power states, 168
data buffers, 34–35
data members (COM), 587
data structures, 24–30
data transfer process, 35–36
  USB devices, 357–358
DBH tool, 19
Dc2 (Device Path Exerciser), 671–672
DDI (device driver interface), 47
  for KMDF, 71
  for UMDF, 69

DDI order rules (KMDF), 847
Deadlock Detection option (Driver Verifier), 678, 680
deadlocks, 41
  Driver Verifier deadlock detection, 45, 678, 680
  WdfDeviceStopIdle method and, 211
debug print statements, converting to ETW, 413, 418
debugger extensions, 14
Debugging Tools for Windows package, 18–19, 699
debugging WDF drivers, 693–694, 697–730. *See also* testing
  annotations. *See* PRE*f*ast for Drivers
  breaking into debugger, 688
  Fx2_Driver sample driver (example), 716–721
  IRQL problems, testing for, 521–523
  Osrusbfx2 sample driver (example), 721–725
  preparing for (KMDF), 711–716
  preparing for (UMDF), 706–710
  reference counts, 591
  system requirements for, 12
  tools for, 18–19, 698–699. *See also specific debugging tool by name*
  tracing. *See* tracing
  using PRE*f*ast. *See* PRE*f*ast for Drivers
  verification information for, 684–686, 689–690
  versioning and, 637
  WDF debugger extensions, 14, 63
  WDF driver installations, 663–666
  WinDbg for, 63, 698, 699–706
    commands, 702
    extensions, 704–706
    installation error debugging, 663
    KMDF debugging session, starting, 714–715
    suggestions for experimenting, 729–730
    viewing KMDF log, 726–729
    viewing trace messages, 725–726
default child list (device property), 145
default I/O callbacks, 289–291
default I/O dispatchers, 311
default I/O queue, 145
default I/O targets, 57, 145, 308
  formatting requests for, 331–334, 341–343
  retrieving, 313
default queue, creating and initializing, 262–263, 265–266
default synchronization scope, 392–393
deferred interrupt processing, 546–547
deferred procedure calls (DPCs), 37, 47, 103, 114, 514–515, 546–547
  deleting, 115, 120
  synchronization scope, 395–396
deferred processing for interrupts, 546–547
DeferredRequestCompleted rule, 849
delay time for WUDFHost, setting, 706–707
Deleted state (I/O targets), 319, 340
DeleteWdfObject method (IWDFObject), 115, 313, 353
deleting objects, 111–112, 115–122
  defined, 112
  KMDF drivers, 115
  memory objects, 447
  UMDF drivers, 113
_deref modifier, 761, 765
dereferencing user-mode pointers, 45
[DestinationDirs] section, INF files, 644
destroy callbacks, 118
destruction of objects, defined, 112
DevCon utility, 637, 658–659, 671
  uninstalling drivers, 662–663
device callback objects. *See also* callbacks
  Skeleton sample driver, 484–486
device characteristics (device property), 145
device descriptors for USB devices, 356–357
device driver interface (DDI), 47
  for KMDF, 71
  for UMDF, 69
device enumeration
  child devices, 143, 157–159
  startup and, 180–183
device hardware resources. *See* hardware resources
device I/O control requests, 33, 48, 225–226, 242
  control codes for, 226–227, 228
  event callbacks for, 282–291
  formatting for I/O targets, 331–333
  internal, 228, 242, 322
  synchronous, 337
device initialization roles (KMDF), 848
device instance ID (device property), 145
device interfaces, 148
  names for KMDF drivers vs., 159
device IRQLs (DIRQLs), 38, 511, 516
  guidelines for running at, 517
  synchronized processing at, 547
  synchronizing, 41
Device Manager, uninstalling drivers with, 662
Device Manager, updating drivers with, 659
device name (device property), 145
device objects, 28–29, 47, 97, 102, 114, 140–147, 152
  callback objects for, 100
  context areas for, 126
  creating, 155
  creation and initialization, 149–157
  deleting, 115, 120
  initializing, 146–147
  in object hierarchy, 111–113
  pointers to, I/O targets vs., 318
  properties, 145–146
  types of, 140
Device Path Exerciser (Dc2), 671–672
device power-down, 178–179
  device removal and, 184–187
  self-managed I/O during, 300–301

device power states, 168–169
device property stores, 99, 447–450
device registers, mapping, 530
device removal
  power-down and, 184–187
  self-managed I/O during, 300–301
  surprise, 178
device restart, 178–179
  device enumeration and, 180–183
  self-managed I/O during, 300
device scope (synchronization), 387, 393
device shutdown. *See* device power-down
device stack, 28–29, 47
  I/O request flow through, 234–236, 308. *See also* default I/O targets
  IRP management, 33–34
  power states, 176–177
  UMDF infrastructure, 77–78
device startup, 178–179
  device enumeration and, 180–183
  self-managed I/O during, 300
device state (device property), 145
device support, 85–89
device tree, 29–30
device uninstallation, 660–661
Device.cpp and Device.h files, 477
DeviceInitAllocate rule, 848
DeviceInitAPI rule, 848
DeviceIoControl function, 33, 225
devnode, defined, 47
diagnosing with software tracing. *See* tracing
DIFxApp (Driver Install Frameworks for Applications), 636, 657–658
  uninstalling drivers, 662
digital certificates, 654
digitally signing driver packages, 653–655
direct I/O, 34–35, 228, 230
  retrieving buffers in KMDF drivers, 249
direct memory access (DMA), 549–581
  basics and terminology, 550–552
  best practices, 581
  device information, 552–555
  implementing DMA drivers, 564–577
  testing drivers, 577–581, 668–669
  Windows DMA abstraction, 555–563
direction of transfer, 355
directives in INF files, 643–644
DIRQLs (device IQLs), 38, 511, 516
  guidelines for running at, 517
  synchronized processing at, 547
  synchronizing, 41
Dirs file, 621
Disable_Enable_With_IO.wsf script, 676
disconnecting interrupts, 540
disk I/O path, 43
Disk Integrity Checking option (Driver Verifier), 678
DISPATCH_LEVEL IRQLs, 38, 392, 396, 511–512, 514–515
  guidelines for running at, 517
  object cleanup, 121
  synchronizing, 41
dispatch types for I/O queues, 56, 259, 385–386
  device scope and (synchronization), 388–389
disposal of objects, 112
distributing driver packages, 655
.dll files, 623
DLL infrastructure for UMDF drivers, 478–480, 600–606
DLLDEF macro, 628
DLLENTRY macro, 628
DllGetClassObject function, 130–132, 479–480, 585, 600, 602–603
DllMain function, 130–132, 478–479, 585, 600–601
  debugging, 708
Dllsup.cpp file, 476
DMA (direct memory access), 549–581
  basics and terminology, 550–552
  best practices, 581
  device information, 552–555
  implementing DMA drivers, 564–577
  testing drivers, 577–581, 668–669
  Windows DMA abstraction, 555–563
DMA chaining. *See* scatter/gather technique
DMA common buffer objects, 102, 114, 147
  deleting, 115, 121
!dma debugger extension, 579
DMA design type, 553
DMA enabler objects, 103, 114, 147, 566–567
  deleting, 115
DMA transaction objects, 103, 114, 147, 550, 567–569
  deleting, 115
DMA transfers, 550
  flushing caches for, 557
  maximum transfer length, 554
DMA Verification option (Driver Verifier), 678, 680
DO_DEVICE_INITIALIZING annotation, 809
DO_POWER_PAGEABLE flag, 175
documentation, debugging tools, 19
documentation, WDK, 14–15
DoTraceMessage macro, 417–418
DoubleCompletion rule, 849
DoubleCompletionLocal rule, 849
Down device object (UMDF device stack), 79, 81
downloading. *See* obtaining (downloading)
DPCs (deferred procedure calls), 37, 47, 103, 114, 514–515, 546–547
  deleting, 115, 120
  synchronization scope, 395–396
DPInst application, 636, 657–658
Drain method (IWDFIoQueue), 260

DrainSynchronously, 260
driver annotations, 774–778
  for __try statement functions, 790
  conditional, 779–783
  constant and non-constant parameter, 788
  diagnostic, 789
  DO_DEVICE_INITIALIZING, 809
  for error messages, 790
  examples, 810–812
  floating-point, 801–802
  format string, 788–789
  for function results, 784–785
  function type class, 799–801
  for interlocked operands, 809–810
  IRQL, 802–809
  memory, 790–792
  nonmemory, 792–799
  Osrusbfx2 sample driver (example), 818–822
  pointing, 787–788
  type-matching, 785–787
driver architecture, 27–28
driver binary files, 623, 642, 661
driver callback objects. *See also* callbacks
  Skeleton sample driver, 481–484
driver code. *See* source code
driver crash, debugging, 710
driver-created objects, 70
  context areas for, 124, 126
  deleting, 116
  file objects, 98
  I/O requests
    deleting, 118
    memory objects and buffers for, 323–330
  parents of, 113
driver-created subrequests, 408–410
driver development vocabulary, 47–49
Driver Hand Verification option (Driver Verifier), 678
Driver Install Frameworks for Applications (DIFxApp), 636, 657–658
driver load, debugging, 706–708, 715
driver manager, 78
driver objects, 97, 135–140
  callback objects for, 100
  deleting, 116
  as root objects in hierarchy, 110, 112–113, 114
Driver Package Installer (DPInst), 636, 657–658
driver packages, defined, 47. *See also* installing WDF drivers
driver projects, how to build, 622–624
driver-specific coding practice errors, detecting, 733
driver stack, 28
driver store, deleting packages from, 661
driver structure
  device interfaces, 148
    names for KMDF drivers vs., 159
  objects. *See* device objects; driver objects
  required components, 129–135
driver testing. *See* testing WDF drivers
driver threads and work items, 523–527
driver tracing, 411–439, 699
  basics of, 412–417
  best practices, 439
  how to support, 420–432
  message functions and macros, 417–419
  running a trace session, 433–439
  tools for, 63, 432–433
  viewing messages with WinDbg, 725–726
driver verification. *See* testing WDF drivers; verifying drivers
Driver Verifier, 46, 677–686. *See also* testing WDF drivers
  deadlock detection, 45, 678, 680
  examples, 679–684
  Force IRQL Checking option, 45, 678, 680, 683–684
  IRQL problems, testing for, 523
  testing DMA drivers, 578–579
driver versioning, 63–64, 637
DriverCLSID directive, 648
Driver.cpp and Driver.h files, 477
DriverCreate rule, 847
DriverEntry function, 133–134, 137–140
  thread context, 510
drivers, defined, 24–30
/drivers parameter (Driver Verifier), 680
__drv annotations, 774–778
  for __try statement functions, 790
  basic, 777–779
  conditional, 779–783
  constant and non-constant parameter, 788
  diagnostic, 789
  DO_DEVICE_INITIALIZING, 809
  for error messages, 790
  examples, 810–812
  floating-point, 801–802
  format string, 788–789
  for function results, 784–785
  function type class, 799–801
  for interlocked operands, 809–810
  IRQL, 802–809
  memory, 790–792
  nonmemory, 792–799
  Osrusbfx2 sample driver (example), 818–822
  pointing, 787–788
  type-matching, 785–787
!dumpobjects debugger extension, 710
dynamic enumeration of child devices, 143
  in bus drivers, 157–158
dynamic verification tools, 669. *See also* Driver Verifier
  Application Verifier, 693–694
  KMDF Verifier, 45, 687–690
  UMDF Verifier, 690–693

E

_ecount annotation, 761, 766–768
elevated privileges, working with, 15
ENABLE_EVENT_TRACING macro, 632
encapsulation (COM), 587
endpoints on USB devices, 355–356
Enhanced I/O Verification option (Driver Verifier), 678
entry actions, SDV rules for, 826
EnumDevices.wsf script, 676
enumerated types, incorrect, 747–748, 785–787
enumeration of devices
  child devices, 143, 157–159
  startup and, 180–183
enumeration of pipes for USB interfaces, 367
error messages, annotations for, 790
errors
  coding, detecting. *See* PRE*f*ast for Drivers
  detecting in drivers. *See* testing WDF drivers
  driver installation, 664–666
  HRESULT type vs., 594
  IRQL-related, 521–523
  in KMDF drivers, 59, 83
  observing with SDV. *See* SDV (Static Driver Verifier)
  reported in wrong places, 744–746
  in UMDF drivers, 58, 80
  UMDF Verifier, 691
  Windows Error Reporting (WER), 692–693, 696
ES scenario (PwrTest), 675
ETW (Event Tracing for Windows), 411, 416–417
event callbacks. *See* callbacks
events (framework objects), 54
events (WDF), 93. *See also* callbacks
  KMDF naming conventions, 105
  UMDF naming conventions, 96–97
events for driver developers, 22
events of COM objects, 594–595
EvtChildListCreateDevice callback, 158, 183
EvtChildListScanForChildren callback, 182, 183
EvtCleanupCallback callback, 108, 117, 119–121, 139, 185
  cleaning up tracing, 430–431
EvtCleanupContext callback, 186, 189
EvtDestroyCallback callback, 108, 118, 119, 121–122, 185
EvtDestroyContext callback, 186, 189
EvtDeviceArmWakeFromS0 callback, 173, 185, 217
EvtDeviceArmWakeFromSx callback, 173, 185, 214–215, 217, 221
EvtDeviceD0Entry callback, 172, 182, 183, 204–205
  post-interrupt processing, 543
EvtDeviceD0EntryPostInterruptsEnabled callback, 173, 182, 183, 195, 538, 541
  post-interrupt processing, 543
EvtDeviceD0Exit callback, 172, 185, 186, 189, 196, 204–205
  post-interrupt processing, 543
EvtDeviceD0ExitPreInterruptsDisabled callback, 173, 185, 196, 538, 542
  post-interrupt processing, 543
EvtDeviceDisableWakeAtBus callback, 159, 173, 183, 186
EvtDeviceDisarmWakeFromS0 callback, 173, 182, 183
EvtDeviceDisarmWakeFromSx callback, 173, 182, 183, 215
EvtDeviceEject callback, 159
EvtDeviceEnableWakeAtBus callback, 159, 173, 183, 186
EvtDeviceFileCreate callback, 274, 279–280, 391
  execution level, 397
EvtDeviceFilterAddResourceRequirements callback, 173, 182, 183
EvtDeviceFilterRemoveResourceRequirements callback, 173, 182, 183
EvtDevicePrepareHardware callback, 172, 182, 183, 203, 530–531, 534
  idle characteristics configuration, 208–209
EvtDeviceQueryRemove callback, 172
EvtDeviceQueryStop callback, 172
EvtDeviceReleaseHardware callback, 172, 185, 186, 189, 203, 530–531, 536
EvtDeviceRemoveAddedResources callback, 153, 173, 182, 183
EvtDeviceResourceRequirementsQuery callback, 159, 173, 183
EvtDeviceResourcesQuery callback, 159, 173, 183
EvtDeviceSelfManagedIoXxx callbacks, 172, 182–186, 189, 299, 300, 301
  watchdog timer example (KMDF), 301–305
EvtDeviceSetLock callback, 159
EvtDeviceSurpriseRemoval callback, 172, 188
EvtDeviceWakeFromS0Triggered callback, 173, 215
EvtDeviceWakeFromSxTriggered callback, 173, 215, 221
EvtDeviceWdmIrpPreprocess callback, 504
EvtDmaEnablerDisable callback, 186, 189
EvtDmaEnablerEnable callback, 182, 183
EvtDmaEnablerFill callback, 182, 183
EvtDmaEnablerFlush callback, 185, 186, 189
EvtDmaEnablerSelfManagedIoStart callback, 182, 183
EvtDmaEnablerSelfManagedIoStop callback, 185, 186, 189
EvtDpcFunc callback, 391, 515
EvtDriverDeviceAdd callback, 133–135, 138, 146, 152–153, 182, 183, 194
  DMA driver implementation, 564–566
  idle characteristics configuration, 208–209
  processing unsupported I/O requests, 504
  software-only drivers, 192–194
  tasks of, 155
  thread context, 510
EvtDriverUnload callback, 134, 138

EvtFileCleanup callback, 281, 391
  execution level, 397
EvtFileClose callback, 281, 391
  execution level, 397
EvtInterruptDisable callback, 173, 185, 516, 538, 541
  post-interrupt processing, 543
EvtInterruptDpc callback, 391, 404–407, 515, 516, 519, 521, 538, 546–547
  DMA drivers, 575–577
EvtInterruptEnable callback, 173, 182, 183, 516, 538, 541–542
  post-interrupt processing, 543
EvtInterruptIsr callback, 400, 519, 521, 538, 544–545
  DMA drivers, 575–577
EvtInterruptSynchronize method, 516, 546–547
EvtIoCanceledOnQueue callback, 252, 293, 391
EvtIoDefault callback, 283, 286, 289–290, 391
EvtIoDeviceControl callback, 286, 288–289, 391
EvtIoInCallerContext callback, 250–252, 511
EvtIoInternalDeviceControl callback, 286, 391
EvtIoQueueState callback, 268, 391
EvtIoRead callback, 283, 285–286, 391
EvtIoResume callback, 182, 183, 258, 391
EvtIoStop callback, 185, 186, 189, 204, 258, 391
EvtIoTargetXxx callbacks, 320–321
EvtIoWrite callback, 283, 391
EvtObjectCleanup callback, 252
EvtProgramDma callback, 572–575
EvtRequestCancel callback, 391, 405–406
EvtTimerFunc callback, 391, 460, 461–462
EvtUsbTargetPipeReadComplete callback, 375
EvtUsbTargetPipeReadersFailed callback, 375
EvtWmiInstanceExecuteMethod callback, 470
EvtWmiInstanceQueryInstance callback, 470–471
EvtWmiInstanceSetInstance callback, 470, 472, 473
EvtWmiInstanceSetItem callback, 470, 472–473
EvtWorkItem callback, 391, 524–525
ExAcquireResourceExclusiveLite function, 780
ExAllocatePoolWithTag DDI, 442
execution levels in KMDF drivers, 396–398
execution threads. *See* threads
ExecutionLevel attribute, 108
ExInterlockedXxx routines, 500
exit actions, SDV rules for, 826
Exports.def file, 477, 488
extensions, debugger, 704–706. *See also* debugging WDF drivers

F

Fail_Driver3 sample driver (example), 855–859
failure codes, 58–59
fast mutex objects, 41
fatal errors
  during installation, 665
  in KMDF drivers, 83
  in UMDF drivers, 80
faults, memory. *See* page faults
/faults parameter (Driver Verifier), 682–683
FDODriver rule, 853
FDOPowerPolicyOwnerAPI rule, 854
FDOs (functional device objects), 29, 140, 141
  device initialization for (KMDF), 153–154
  power management, 171, 174
    device enumeration and startup sequence, 182
    power-down and removal sequence, 185
    software-only drivers, 190
file handle I/O targets (UMDF), 353–354
file objects, 98, 103, 114
  callback objects for, 100
  context data for, 123
  deleting, 116
  with I/O requests, 271–272
  intra-stack files and, 313
  in object hierarchy, 112
file system drivers, 89
FILE_AUTOGENERATED_DEVICE_NAME property, 159
FileHandle dispatchers, 312
FileObjectConfigured rule, 847
filter (device property), 145
filter DOs (device objects), 29, 140, 141
  device initialization for (KMDF), 154
  power management, 175
    software-only drivers, 190
filter drivers, 29, 141, 144
  I/O request flow through, 235, 237
  as power policy owners, 190
filtering PRE*f*ast results, 815
fixed device names, 159
fixed-size buffer annotations, 766–768
/flags parameter (Driver Verifier), 680–681
floating-point annotations, 733, 801–802
flushing caches for DMA transfers, 557
Force IRQL Checking option (Driver Verifier), 678, 680, 683–684
Force Pending I/O Requests option (Driver Verifier), 678
FormatRequestForControlTransfer method (IWDFUsbIoTarget), 370
FormatRequestForIoctl method (IWDFIoTarget), 331
FormatRequestForRead method (IWDFIoTarget), 331
FormatRequestForWrite method (IWFDIoTarget), 284, 332, 333
formatting I/O requests, 330–336
FormatUsingCurrentType method (IWDFIoRequest), 331
FormatUsingCurrentType method (IWDFRequest), 342
forwarding of I/O requests (automatic), 272–273
framework-created framework objects (UMDF), 70

framework objects, 52–55, 75, 104, 638–639
context areas for, 124–125
deleting, 115–117
hierarchy of, 54–55
I/O request flow, 239–241
with KMDF, 71–72
with UMDF, 69–70
framework selection, 89
framework startup and shutdown, 178–179
free builds of co-installers, 640
free builds of drivers, 620
debugging with WinDbg, 700
free builds of Windows, obtaining, 18, 47
freeing memory, annotations for, 790–791
_full annotation, 761
fully-powered working state, 168
function drivers, 29, 141, 144
create requests, not accepting, 279–280
I/O request flow through, 235, 237
interrupt handling, 538
function parameters, annotations on, 754–756
function pointers, incorrect use of, 750
function result annotations, 784–785
function return values, checking, 773–774
function role type declarations (SDV), 832–834, 860–862
function type class mismatches, 746–747
function typedef declarations, annotations on, 758–760
function usage errors, detecting with PRE*f*ast, 733
functional device object (FDO), 29, 140, 141
device initialization for (KMDF), 153–154
power management, 171, 174
device enumeration and startup sequence, 182
power-down and removal sequence, 185
software-only drivers, 190
functions, annotations on, 754–756
Fx2_Driver sample driver, 16
debugging, 716–721
INF file for, 649–651
obtaining (downloading), 16

G

general I/O targets, 309–310
general objects (base objects), 97, 454–455
deleting, 116
in object hierarchy, 112
general-purpose annotations, 760–774
generic objects, 103, 114
deleting, 116
Get functions, 96, 104
GetCompletionParams method (IWDFIoRequest), 371
GetCompletionStatus method (IWDFRequestCompletionParams), 371
GetDataBuffer method (IWDFMemory), 246, 444
GetDefaultIoTarget method (IWDFDevice), 313, 341
GetDeviceControlTransferParameters method (IWDFUsbCompletionParams), 371
GetInputMemory method (IWDFIoRequest), 245–246
splitting I/O requests, 347
GetNameAt method (IWDFNamedPropertyStore), 448
GetNameCount method (IWDFNamedPropertyStore), 448
GetNamedValue method (IWDFNamedPropertyStore), 448, 449
GetNumEndPoints method (IWDFUsbInterface), 363
GetNumInterfaces method (IWDFUsbTargetDevice), 363
GetOutputMemory method (IWDFIoRequest), 245–247
splitting I/O requests, 347
GetPnpCapability method (IWDFDeviceInitialize), 150
GetSize method (IWDFMemory), 444
GetState method (IWDFIoQueue), 260
GetState method (IWDFIoTargetStateManagement), 319
GetType method (IWDFUsbTargetPipe), 364
global nonmemory resources, annotations for, 794–795
glossary for driver development, 47–49
guards (SDV rules), 826–827
GUIDs (globally unique identifiers), 592
for device interface classes, 148
for driver tracing, 425–427

H

hardware requirements for driver development, 12
hardware resources, 529–537
hardware scatter/gather technique. *See* scatter/gather technique
heap (memory), 43, 48
hibernation files, 174–176
hierarchy of framework objects, 54–55
hierarchy of WDF objects, 94, 110–122, 384
KMDF drivers, 112–114, 119
power management, 173
synchronization scope, 394–396
UMDF drivers, 114–115
highest-powered state, defined, 168
HIGH_LEVEL IRQLs, 512, 516
guidelines for running at, 517
holding nonmemory resources, annotations for, 797
host controllers, 355
HRESULT type, 58, 593–594
hybrid DMA devices, 551–552, 553
hyperthreading support, 12

I

I/O dispatchers, 310–312
I/O manager (Windows system), 27

I/O mapping, 530
I/O processing, 221
adaptive time-outs (UMDF), 297
completion routines (I/O), 34, 47, 238–239. *See also* completing I/O requests
DMA drivers, 575–577
for I/O requests, 334–336
control requests. *See* IOCTL (I/O control) requests
event callbacks, 271–291
automatic forwarding, 272–273
for cleanup and close requests, 280–281
for create requests, 273–280
file object creation, 271–272
for read/write and IOCTL requests, 282–291
self-managed, 298
serialization, synchronization scope and, 387–398
queue objects, 56, 98, 130, 147, 253–271
callback objects for, 100
configuration and request types, 254–256
control of, 259–261
creating (KMDF example), 265–267
creating (UMDF example), 261–264
deleting, 116
dispatch types (flow control), 56, 259, 385–386, 388–389
manual, retrieving requests from, 267–271
in object hierarchy, 112–113
power management for, 171, 196–197, 257–258
request completion, 291–293
adaptive time-outs (UMDF), 297
parent objects and, 121
request completion information objects, 98
request completion parameters objects, 98–99
USB targets, 99
request flow, 231–243
through KMDF drivers, 236–237, 309
through UMDF device stack, 234–236, 308
request handlers, 239, 242–243
request objects, 32–33, 98, 243, 322–353
buffers and memory objects, 244–252
callback objects for, 100, 123
canceled. *See* canceled I/O requests
completing. *See* completing I/O requests
context data for, 123
creating, 322–323, 346–350
deleting, 112, 116, 118
dispatching with queue objets, 56
file objects created for, 271–272
formatting, 330–336
how to send, 337–353, 376–377
to KMDF drivers, 84
lifetimes, 252–253
memory objects and buffers for, 323–330
in object hierarchy, 112–113
suspension, 293, 296
to UMDF drivers, 80–81
request types, 224–228
requests, unsupported, 503–506
self-managed, 297–305
synchronization. *See* synchronization
target objects, 57, 98, 147, 307–321
creation and management, 313–321
deleting, 116
UMDF file handle targets, 353–354
UMDF implementation, 310–313
USB targets, 99, 355–376
transfer types, 34–35, 228–231
WDF model, 55–59, 75
Windows model, 31–36
completion processing, 239
I/O target files, 310, 312–313
I/O Verification option (Driver Verifier), 678, 680
IClassFactory interface, 480–481, 603–606
CreateInstance method, 132, 135–137, 480–481, 603, 604–605
Identification impersonation level, 276
IDL (interface definition language), 595–597
idle devices, 174
idle support for KMDF drivers, 207–212, 220
example, 217–220
IDriverEntry interface, 100, 130, 132, 135, 482–483, 610–611
OnDeviceAdd method, 132, 136, 149, 181, 194, 483–484, 598
!IFDEF/!ENDIF macro, 632
IFileCallbackCleanup interface, 100, 281
canceling all I/O requests for files, 351–352
IFileCallbackClose interface, 100, 281
IIDs (interface IDs), 592
IImpersonateCallback interface, 100, 276–277
Impersonate method (IWDFIoRequest), 276–277
impersonation (UMDF drivers), 275–279
implicit order of evaluation, detecting, 742–743
__in annotation, 761–764
IN parameter, 590, 764
INCLUDES macro, 627, 631
INF File Syntax Checker (ChkINF), 671
.inf files (INFs), 48, 477, 489–493, 621, 643
creating, 643–648
duplicate directives in, 665
examples of, 648–653
INF_NAME macro, 631
INFO scenario (PwrTest), 675
inheritance, 587
execution levels, 397
synchronization scope, 394–396
Initialize public helper method, 484–485
initializing device objects, 146–147, 149–157
initializing tracing, 427–429
inline assembler, PRE*f*ast and, 750

__inout annotation, 761–764
INOUT parameter, 591, 764
input buffer naming, 247
input-output model for Windows, 31–36
  completion processing, 239
input parameter annotations, 761–764
InputBufferAPI rule, 851
installing WDF drivers, 635–666
  basics of, 636–637
  considerations, 637–642
  INFs, creating, 643–648
  INFs, examples of, 648–653
  methods for, 655–663
  package components, 642–643
  signing and distributing packages, 653–655
  troubleshooting, 663–666
installing WDK, 11–14
intellectual property in trace messages, 413
interface definition language (IDL), 595–597
interface IDs (IIDs), 592
interface pointers as method parameters, 590
interfaces (COM), 587–589
interfaces (WDF)
  UMDF framework, 97–101
  UMDF naming conventions, 96
interlocked operands, annotations for, 809–810
InterlockedXxx routines, 500
  InterlockedIncrement, 609
intermediate power states, 168
internal device I/O control requests, 228, 242
  creating, 322
  formatting for I/O targets, 332–333
  synchronous, 337
internal request handlers, 241
internal trace logger (KMDF), 669–670
Internal.h file, 477
interrupt objects, 103, 114, 147, 538–541
  deleting, 116
interrupt request levels. *See* IRQLs (interrupt request levels)
interrupt service routines. *See* ISRs
interrupt spin locks, 48, 400. *See also* spin lock objects
interrupt storms, 543
interrupt transfers (I/O model), 357–358
interruption, thread. *See* threads
interrupts, 37–39, 48
  in data transfer process, 35–36
interrupts and interrupt handling, 537–547. *See also* IRQLs (interrupt request levels)
  deferred processing, 546–547
  enabling and disabling interrupts, 541–542
  ISRs (interrupt service routines), 48, 544–547
    synchronizing, 41
  post-interrupt processing, 543
  synchronized processing at DIRQL, 547
intra-stack files for I/O targets, 312–313
_INX directive (Makefile.inc), 633
IObjectCleanup interface, 100, 117, 124–125, 454
IOCTL (I/O control) requests, 33, 48, 225–226, 242
  control codes for, 226–227, 228
  event callbacks for, 282–291
  formatting for I/O targets, 331–333, 332–333
  internal, 228, 242, 322
  synchronous, 337
IoGetDeviceProperty function, 146
IoRegisterPlugPlayNotification function, 503
IoSetCompletionRoutine function, 504–505
IoSetCompletionRoutineEx function, 504–505
IoWmiXxx functions, 442
IPI_LEVEL IRQLs, 512
IPnpCallback interface, 100, 184, 188, 198, 200–201
  OnD0Entry method, 172, 181, 194, 198, 200–201
  OnD0Exit method, 172, 188, 196, 198, 200–201
  OnQueryRemove method, 172, 198
  OnQueryStop method, 172, 198
  OnSurpriseRemoval method, 172, 187, 198, 200–201
  Plug and Play and power callbacks, 172
  Skeleton sample driver with, 485
IPnpCallbackHardware interface, 100, 184, 197, 199–200
  OnPrepareHardware method, 172, 181, 197, 199
  OnReleaseHardware method, 172, 184, 188, 197, 200
  Plug and Play and power callbacks, 172
  Skeleton sample driver with, 485
IPnpCallbackSelfManagedIo interface, 100, 184, 188, 298–299
  Plug and Play and power callbacks, 172
  Skeleton sample driver with, 485
IQueueCallbackCreate interface, 100, 274–275
IQueueCallbackDefaultIoControl interface, 100
IQueueCallbackDefaultIoHandler interface, 100, 263, 274, 283, 286, 289
IQueueCallbackDeviceIoControl interface, 286
  OnDeviceIoControl method, 287–288, 390
IQueueCallbackIoResume interface, 100, 258
IQueueCallbackIoStop interface, 100, 188, 258
  OnIoStop method, 184, 188, 295, 390
IQueueCallbackRead interface, 100, 263, 283
IQueueCallbackRequestCompletion interface, 263
IQueueCallbackState interface, 100
IQueueCallbackStateChange interface, 268
IQueueCallbackWrite interface, 100, 263, 283–284
IRequestCallbackCancel interface, 100
IRequestCallbackRequestCompletion interface, 100, 263, 334
  OnCompletion method, 343, 390
IRP Logging option (Driver Verifier), 678
IRPs (I/O request packets), 31, 33–34, 48, 227–228
  IRP_MJ_CLEANUP code, 227, 241, 242
  IRP_MJ_CLOSE code, 227, 241, 242

IRPs (I/O request packets) (*continued*)
IRP_MJ_CREATE code, 227, 241, 242
IRP_MJ_DEVICE_CONTROL code, 227, 241, 242
IRP_MJ_INTERNAL_DEVICE_CONTROL code, 228, 241, 242
IRP_MJ_PNP code, 228
IRP_MJ_POWER code, 228
IRP_MJ_READ code, 228, 241, 242
IRP_MJ_SHUTDOWN code, 228
IRP_MJ_SYSTEM_CONTROL code, 228, 464–465
IRP_MJ_WRITE code, 228, 241, 242
preprocessing, 241
request flow, 233
transfer types and, 229–230
types of information in, 34
!irql debugger command, 522
IRQLs (interrupt request levels), 37–38, 48, 511–518. *See also entries at* interrupt
annotations, 802–809
incorrect, calling functions as, 743–744
interruption scenarios, 518–523
page faults and, 42
processor- and thread-specific, 512–517
IsEqualIID utility, 610
isFunctionClass$ function, 783
IsInEndPoint method (IWDFUsbTargetPipe), 364
isochronous transfers (I/O model), 357–358
IsOutEndPoint method (IWDFUsbTargetPipe), 364
ISRs (interrupt service routines), 48, 544–547
synchronizing, 41
IUnknown interface, 99, 480, 482, 587, 589
how to implement, 608–611
IWDFDevice interface, 69, 97
ConfigureRequestDispatching method, 263–264
CreateDeviceInterface method, 152
CreateIoQueue method, 197, 198, 261
CreateRequest method, 322, 598–599
GetDefaultIoTarget method, 313
IWDFDeviceInitialize interface, 146, 149–151
AutoForwardCreateCleanupClose method, 150, 272, 274, 281
SetFilter method, 150, 151, 192
SetLockingConstraint method, 150, 151, 390
SetPowerPolicyOwnership method, 150, 151, 192
IWDFDriver interface, 97
CrateWdfMemory method, 327
CreatePreallocatedWdfMemory method, 327, 329, 443–444
CreateWdfMemory method, 443
IWDFDriverCreatedFile interface, 98
IWDFFile interface, 98
intra-stack files for I/O targets, 312–313
IWDFFileHandleTargetFactory interface, 97, 353–354
IWDFIoQueue interface, 98, 260, 268
ConfigureRequestDispatching method, 263–264
IWDFIoRequest interface, 69, 98, 292, 371
CancelSentRequest and CancelSentRequestsForFile method, 351
FormatUsingCurrentType method, 331
MarkCancelable, 294
memory object methods, 245–246
Send method, 337, 343
SetCompletionCallback method, 334
UnmarkCancelable, 294
IWDFIoTarget interface, 98, 331–332
IWDFIoTargetStateManagement interface, 98, 319–320
IWDFMemory interface, 98, 245–247, 443–444
IWDFNamedPropertyStore interface, 99, 448
IWDFObject interface, 97
AcquireLock and ReleaseLock methods, 386
CreateWdfFile method, 312
DeleteWdfObject method, 115, 313, 353
IWDFRequestCompletionParams interface, 98, 336, 371
IWDFTargetPipe interface, 99
IWDFUsbCompletionParams interface, 371
IWDFUsbInterface interface, 99, 359, 361, 363
IWDFUsbRequestCompletionParams interface, 99
IWDFUsbTargetDevice interface, 99, 359
IWDFUsbTargetFactory interface, 359, 362
IWDFUsbTargetPipe interface, 359

K

KD debugger, 699
KeAcquireFastMutex function, 514
KeAreApcsDisabled function, 514, 522
KeEnterCriticalRegion function, 513
KeGetCurrentIrql function, 522
KeLeaveCriticalRegion function, 513
KeLowerIrql function, 513
KePulseEvent function, 509
KeReleaseFastMutex function, 514
KeReleaseSemaphore function, 509
kernel, about, 24
kernel dispatcher objects, 41, 48
__kernel_driver analysis mode (PRE*f*ast), 734
kernel-mode debugging, 698, 711–713
kernel-mode framework. *See* KMDF drivers
kernel-mode programming, 26–27, 36–44, 500–503. *See also* WDM drivers
coding practice errors, detecting, 733
threads, 508. *See also* threads
tips for, 44–46
kernel mode (Windows system), 26
kernel objects, 24–30, 48
kernel stack, 43–44. *See also* memory
kernel subsystems (WDF architecture), 74
kernel subsystems (Windows architecture), 27
KernRate utility, 672–673
KernRate Viewer (KRView), 672–673

KeSetEvent function, 509
KeSynchronizeExecution function, 523
KetGetCurrentIrql function, 514
KMDF debugger extensions, 580–581, 705
KMDF driver development, system requirements for, 12
KMDF drivers, 67–68, 71–73, 83, 171. *See also* WDF driver model
  callbacks, 54, 94. *See also* callbacks
  child device enumeration, 157–159
  choosing between UMDF and, 89
  collection objects, 102, 114, 455–459
    deleting, 115
  debugging, system requirements for, 12
  device and driver support, 88–89
  device naming techniques, 159–162
  device object creation and initialization, 152–157
  DMA support. *See* DMA
  driver object creation, 137–140
  errors in, 59, 83
  how to build, 622–624
  I/O. *See* I/O processing
  installing, 635–666
    basics of, 636–637
    co-installer package, 641, 642
    considerations, 637–642
    INFs, creating, 643–648
    INFs, examples of, 648–653
    methods for, 655–663
    package components, 642–643
    signing and distributing packages, 653–655
    troubleshooting, 663–666
  kernel-mode driver support routines, 500–503
  object attributes, 94
  power management
    device enumeration and startup sequence, 182, 183
    idle support, 207–212, 217–220
    power-down and removal sequence, 185, 186
    power-pageable devices, 174–176
    power states, 169
    simple hardware function driver (example), 202–205
    software-only filter driver (example), 192–194
    surprise-removal sequence, 189
    wake support for KMDF drivers, 174, 207, 212–217, 217–222
  registry objects and methods, 450–453
  sample driver. *See* Osrusbfx2 sample driver
  structure and requirements, 151–155
  threads. *See* threads
  timer objects, 104, 114, 459–464
    deleting, 116, 120
    synchronization scope, 395–396
  typical I/O requests to, 84
  WDF debugger extension, 63
  WMI support in, 464–473
KMDF infrastructure, 81–84
KMDF log, viewing, 726–729
KMDF object model, 72, 75, 102–105
  context areas, 126–128
  object creation, 107–110
  object hierarchy, 114–115
  object lifetime, 115–122
KMDF Verifier, 45, 687–690
KMDF_VERSION macro, 631
KmdfService directive, 647
KRView (KernRate Viewer), 672–673

L

language independence of COM, 586
legacy device drivers, 143–144
libraries, including in SDV verification, 836–837
libraries for WDK, 14
life cycle testing, 696
lifetime
  buffers, 324–325
  COM (Component Object Model), 587, 589–590, 600
  objects, 94, 110–112
    callback objects, 113
    KMDF drivers, 115–122
    request, memory, and buffer pointer objects, 252–253
    UMDF drivers, 115–119
line-based interrupts, 537–538
_LNG directive (Makefile.inc), 633
local storage (memory allocation), 442–443
Locals window (WinDbg), 701
LocalService account, 62
LocalService security context, 275
lock objects, 147
locking constraint. *See* synchronization model (device property); synchronization, scope of
LockServer method (IClassFactory), 480, 481, 603, 606
log, trace. *See* tracing
Logman utility, 432
lookaside list objects, 103, 114, 445
  deleting, 116, 120
low-memory conditions, preparing for, 44
low-memory simulation, 688
Low Resources Simulation option (Driver Verifier), 678, 682–683
lower device stack (UMDF), 79, 81
lower edge, WDF frameworks, 76
lowering IRQL, annotations for, 805

M

macroDefined$ function, 783
macros, 46
macros for debugging, 699
maintaining IRQL, annotations for, 806

maintaining released drivers, 696
major version numbers, 63–64, 637
  updates, 638
Make file, 621
  Fx2_Driver sample driver, 629
  Osrusbfx2 sample driver, 632–633
  Skeleton sample driver, 477, 488
Makefile.inc file, 621–622
  Fx2_Driver sample driver, 629
  Osrusbfx2 sample driver, 632–633
  Skeleton sample driver, 477, 488
manual I/O request dispatching, 56, 259, 264, 267–271, 385
[Manufacturer] section, INF files, 644, 646
map registers (DMA model), 557–560
  scatter/gather support, 560–562
  transfers to physical memory, 562–563
mapping hardware resources, 534–537
MarkCancelable (IWDFIoRequest), 294
MarkCancOnCancReq rule, 850
MarkCancOnCancReqLocal rule, 850
marking requests as cancelable, 295
maximum IRQL, annotations for, 804–805
maximum transfer length (DMA devices), 554
MDL rules (KMDF), 850–852
MdlAfterReqCompletedXxx rules, 852
MDLs (memory descriptor lists), 44
meetings for driver developers, 22
MemAfterReqCompletedXxx rules, 852
memory, 42–44
  allocating, 44, 442–447
    annotations for, 790–791, 798–799
    tagged memory allocation, 44
  deleting, 116
  direct memory access. *See* DMA
  low-memory simulation, 688
  MDLs (memory descriptor lists), 44, 850–852
  page faults, 42, 45, 48
    Driver Verifier IRQL checking, 45, 678, 680, 683–684
memory annotations, 790–792
memory errors, detecting with PRE*f*ast, 733
memory mapping, 530
memory objects, 98, 103, 114, 244, 247–252
  deleting, 116, 120
  for driver-created I/O requests, 323–330
  KMDF rules for verification, 850, 852
  lifetime of, 252–253
  need for, 245
  in object hierarchy, 112
  parents of, 325–326
Memory Pool Monitor (PoolMon), 674
memory pools, 43, 48
message-based interrupts, 537–538
METHOD_NEITHER I/O, 34–35, 228, 230–231
  parameter validation, 62
  retrieving buffers in KMDF drivers, 248–250
  thread context and, 510–511
methods (framework objects), 54
  with KMDF, 72
  with UMDF, 69
methods (WDF), 92
  KMDF naming conventions, 104
  UMDF naming conventions, 96
Microsoft Hardware Newsletter, 20
Microsoft IDL. *See* IDL
Microsoft Windows Device Testing Framework, 675–676
MIDL. *See* IDL
minimum IRQL, annotations for, 804–805
minor version numbers, 63–64, 637
  updates, 638
Miscellaneous Checks option (Driver Verifier), 678
MISCFILES macro, 627, 631
MmMapIoSpace function, 534, 536
MmUnmapIoSpace function, 537
MOF resources, 466, 622
MSC_WARNING_LEVEL macro, 628
MSDN newsgroups, 21
MSR (Microsoft Research), 825
multiprocessor systems, 12, 519–521
multithreading. *See* threads

N

names
  COM conventions, 588–589
  device objects, 145, 160
  I/O targets, 315
  input and output buffers, 247
  KMDF conventions, 104–105
  KMDF device naming, 159–162
  methods, 93
  property stores, 448–449
  UMDF conventions, 95–97
neither I/O, 34–35, 228, 230–231
  parameter validation, 62
  retrieving buffers in KMDF drivers, 248–250
  thread context and, 510–511
nesting driver annotations, 779, 781
new operator, device callback object creation, 149
newsgroups, 21
NIC_ACK_INTERRUPT macro, 546
NIC_INTERRUPT_Xxx macros, 545
NMake all command, 624
NMI (nonmaskable interrupt) callbacks, 516
no scope (synchronization), 387
noise in coding style, 749–752, 815
non-power-managed queues, 179, 184, 185, 186, 188, 189, 258
non-power-pageable devices, 174–176

noncancelable I/O requests, 294
nondefault queues, creating, 263–264, 266–267
None method (IWDFIoRequest), 292
nonfatal errors, handling, 58–59
NonFDONotPowerPolicyOwnerAPI rule, 854
nonmaskable interrupts, 516
nonmemory resource annotations, 792–799
nonpaged pool (memory), 43, 48
NonPnpDrvPowerPolicyOwnerAPI rule, 854
__notnull annotation, 765
NotPowerPolicyOwner rule, 853–854
_NT_TARGET_VERSION macro, 628
NTDDI_VERSION macro, 628
NTDEV list server, 21
NTFSD list server, 21
NTSTATUS type, 48, 59, 593
NTTARGETFILES macro, 627, 631
__null annotation, 765
NULL pointers, detecting, 741–742, 787
__nullnullterminated annotation, 761, 772
__nullterminated annotation, 757, 761, 771–772

O

object context, 95, 122–128
  KMDF drivers, 126–128
  UMDF drivers, 122–125
object files, 623
object lifetime, 94, 110–112
  callback objects, 113
  KMDF drivers, 115–122
  request, memory, and buffer pointer objects, 252–253
  UMDF drivers, 115–119
object manager, defined, 48
object model for WDF frameworks, 72, 75
object presentation lock, 386
object synchronization lock, 386
objects. *See* kernel objects
objects (COM), 587–589
  how to use, 597–600
  implementing classes for, 607
  lifetime management, 597–600
  properties and events, 594–595
objects (KMDF), 102–104
  context areas, 126–128
  creating, 107–110
  deleting, 115–122
objects (UMDF), 95
  context areas, 122–125
  creating, 106–107
  deleting, 115–119
  tracking, 709–710
objects (WDF), 91–128. *See also* KMDF object model; UMDF object model
  attributes, 94, 108
  context areas, 122–128
  deleting, hierarchy and, 111
  implementation, 95–105
  object creation, 106–110
    device objects, 149–157
  object hierarchy and lifetime, 110–122. *See also* hierarchy of WDF objects
  UMDF framework, 97–101
OBJ_PATH directive (Makefile.inc), 633
obtaining (downloading)
  builds of Windows, 17–18
  Debugging Tools for Windows package, 18–19
  OSR Learning Kits, 19–20
  WDK, 13
  Windows symbol files, 19
off (powered-down) state, 168
OnCancel method (IRequestCallbackCancel), 390
OnCleanup method (IObjectCleanup), 113–114, 125, 390
OnCleanupFile callback (IFileCallbackCleanup), 351
OnCloseFile method (IFileCallbackClose), 390
OnCompletion method (IRequestCallbackRequestCompletion), 343, 390
OnCreateFile method (IQueueCallbackCreate), 274–275, 390
OnD0Entry method (IPnpCallback), 172, 181, 194, 198, 200–201
OnD0Exit method (IPnpCallback), 172, 188, 196, 198, 200–201
OnDefaultIoHandler method (IQueueCallbackDefaultIoHandler), 290, 390
OnDeinitialize method (IDriverEntry), 132, 482
OnDeviceAdd method (IDriverEntry), 132, 136, 149, 181, 194, 483–484, 598
OnDeviceControl method, 391
OnDeviceIoControl method (IQueueCallbackDeviceIoControl), 287–288, 390
OnImpersonate method (IImpersonateCallback), 390
OnImpersonation callback (IImpersonateCallback), 277–278
OnInitialize method (IDriverEntry), 132, 136, 482
OnIoResume method (IQueueCallbackIoResume), 181, 390
OnIoStop method (IQueueCallbackIoStop), 184, 188, 295, 390
OnPrepareHardware method (IPnpCallbackHardware), 172, 181, 197, 199
OnQueryRemove method (IPnpCallback), 172, 198
OnQueryStop method (IPnpCallback), 172, 198
OnRead method (IQueueCalbackRead), 263, 390
OnReleaseHardware method (IPnpCallbackHardware), 172, 184, 188, 197, 200
OnSelfManagedIoCleanup method (IPnpCallbackSelfManagedIo), 184, 188, 299

OnSelfManagedIoFlush method (IPnpCallbackSelfManagedIo), 184, 188, 299
OnSelfManagedIoInit method (IPnpCallbackSelfManagedIo), 181, 188, 299, 300
OnSelfManagedIoRestart method (IPnpCallbackSelfManagedIo), 181, 299
OnSelfManagedIoStop method (IPnpCallbackSelfManagedIo), 299
OnSelfManagedIoSuspend method (IPnpCallbackSelfManagedIo), 184, 299
OnStateChange method (IQueueCallbackStateChange), 390
OnSurpriseRemoval method (IPnpCallback), 172, 187, 198, 200–201
OnWrite method (IQueueCallbackWrite), 283–284, 390
Open Systems Resources, 19–21
_opt modifier, 761, 764
OPTIONAL macro, 764
order of evaluation, implicit, 742–743
ordinary spin locks, 400. *See also* spin lock objects
Oshins, Jake, 171
OSR Learning Kits, obtaining, 19–20
OSR Online, 21
Osrusbfx2 sample driver, 17
  debugging, 721–725
  INF file for, 652
  with PRE*f*ast annotations, 818–822
__out annotation, 761–764
OUT parameter, 590–591, 764
output buffer naming, 247
output files (Build utility), 623–624
output parameter annotations, 761–764
OutputBufferAPI rule, 851

P

packages for installation. *See* installing WDF drivers
packet-based DMA devices, 551, 553
page faults, 42, 45, 48
  Driver Verifier IRQL checking, 45, 678, 680, 683–684
paged pool (memory), 43, 48
PAGED_CODE macro, 522
PAGED_CODE_LOCKED macros, 522
paging files, powering down and, 174–176
parallel I/O request dispatching, 56, 259, 385
parameter validation, 62, 242
parent (device property), 146
parentheses errors, 750
ParentObject attribute, 108
parents of objects, 110. *See also* hierarchy of WDF objects
  completed I/O requests, 121
  driver-created I/O request objects, 113
  memory objects, 325–326
  powering down, 173
  synchronization scope, 394–396
_part annotation, 761
pass-through commands, 143
PASSIVE_LEVEL IRQLs, 38, 392, 396, 511–512, 513–514
  object cleanup, 120–121
  synchronizing, 40
  VERIFY_IS_IRQL_PASSIVE_LEVEL( ) macro, 46
PC_LEVEL IRQLs, 511
PCI devices, 168
PdoDeviceInitAPI rule, 848
PDOs (physical device objects), 29, 140, 142
  enumeration of child devices, 157–159
  power management, 171, 174–175
    device enumeration and startup sequence, 183
    power-down and removal sequence, 186
    software-only drivers, 190
pending I/O requests, 259
performance
  annotations in source code, 752–754
  checked builds, 700
  coding noise, 749–752, 815
  filtering PRE*f*ast results, 815
  WPP trace message execution, 413
periodic callbacks, requesting. *See* timer objects
physical addressing capability (DMA devices), 554
physical device object (PDO), 29, 140, 142
  enumeration of child devices, 157–159
  power management, 171, 174–175
    device enumeration and startup sequence, 183
    power-down and removal sequence, 186
    software-only drivers, 190
physical memory, transfers for. *See* DMA (direct memory access)
pipes, 355. *See also* USB pipe objects
platform-independent drivers, 533–534
Plug and Play, 36, 165–207
  basics of, 167–168
  callback sequences, 176–189
  driver types and device objects, 144–145
  I/O request flow handler, 239
  implementing in WDF drivers, 189–190
  KMDF drivers for, 134
  PnP manager (Windows system), 27, 167
    error codes, 666
    installing drivers with, 636, 656–657
  serialization of callbacks, 385
  simple hardware drivers, 194–207
  software-only drivers, 190–194
  WDF support for, 59–61, 170–176
    defaults, 170
Plug and Play capabilities (device property), 146
Plug and Play CPU Test (PNPCPU), 673–674
Plug and Play Driver Test (Pnpdtest), 673

Plug and Play state (device property), 146
PnP. *See* Plug and Play
PnP manager (Windows system), 27, 167. *See also* Plug and Play
error codes, 666
installing drivers with, 636, 656–657
PnP/Power model (WDF), 75
PNPCPU utility, 673–674
Pnpdtest.exe, 673
policy, power. *See* power policy
policy, PRE*f*ast integration, 815
pool tags, 442
Pool Tracking option (Driver Verifier), 678, 680
PoolMon utility, 674
pools (memory), 43, 48
PORT mapping. *See* I/O mapping
__possibly_notnullterminated annotation, 761, 772–773
power capabilities (device property), 146
power-down (device), 178–179
device removal and, 184–187
self-managed I/O during, 300–301
shutdown sequences, 178
POWER_LEVEL IRQLs, 512
power-managed queues, 171, 179, 181–186, 188–189, 195, 257–258
power management, 36, 167–222
basics of, 168–170
callback sequences, 176–189
I/O request flow handler, 239
idle and wake support (KMDF drivers), 207–222
implementing in WDF drivers, 189–190
policy owner DDI rules, 853–855
queue objects and, 171, 196–197, 257–258
serialization of callbacks, 385
simple hardware drivers, 194–207
software-only drivers, 190–194
WDF support for, 59–61, 170–176
defaults, 170
within Windows system architecture, 27
Power Management Test Tool (PwrTest), 674–675
power-pageable devices, 174–176
power policy, 146, 169–170
SetPowerPolicyOwnership method (IWDFDeviceInitialize), 150, 151, 192
WdfDeviceInitSetPowerPolicyEventCallback, 153, 217, 853
WdfDeviceInitSetPowerPolicyOwnership, 143
power policy ownership (device property), 146
power policy state (device property), 146
power state (device property), 146
power states, 168–169
changes, callback sequence and, 177–178
self-managed I/O and, 298
system power state queries, 221
powered-down (off) state, 168
PPM scenario (PwrTest), 675
#pragma warning directives, 750–751
precedence rules errors, detecting with PRE*f*ast, 733
preconditions for SDV rules, 854–855
preempted threads, 518
PRE*f*ast defect log text output, 740–741
PRE*f*ast defect log viewer, 737–740
PRE*f*ast for Drivers, 46, 670, 731–822
best practices, 815–818
coding practices to improve results, 748–752
driver annotations, 774–778
basics and conventions, 777–779
conditional, 779–783
constant and non-constant parameters, 788
diagnostic annotations, 789
DO_DEVICE_INITIALIZING annotation, 809
examples, 810–812
floating-point annotations, 801–812
format string annotations, 788
function result annotations, 784
function type class annotations, 799–801
for functions in __try statements, 790
for interlocked operands, 809–810
IRQL annotations, 802–809
memory annotations, 790–792
nonmemory resource annotations, 792–799
pointer annotations, 787–788
type-matching annotations, 785–787
general-purpose annotations, 760–774
how to use, 734–748
how to use annotations, 752–760
how to write and debug annotations, 813–815
Osrusbfx2 sample driver (example), 818–822
preferred functions, annotations for, 789
preprocessing IRPs, 241
priority boost (constant), 292, 509
priority, thread, 508
privacy, trace messages and, 413, 417
processor cache, 556–557
processor-specific IRQLs, 512–517
PROFILE_LEVEL IRQLs, 511
program code. *See* source code
programming interface (WDF), 54
projects, how to build, 622–624
properties
COM objects, 594
device objects, 145–146
framework objects, 54, 70, 72
WDF, 92
KMDF naming conventions, 104–105
WDF
UMDF naming conventions, 96
property store objects, 99, 447–450
PsCreateSystemThread function, 523

Purge method (IWDFIoQueue), 260
PurgeSynchronously method (IWDFIoQueue), 260
PVOID as parameter type, 787
PwrTest utility, 674–675

Q

quantum, defined, 508
QueryInterface method (IUnknown), 137, 589, 598, 599–600, 610–611
queue dispatch types, 56, 259, 385–386
  device scope and (synchronization), 388–389
queue objects, 56, 98, 130, 147, 253–271
  callback objects for, 100
  configuration and request types, 254–256
  control of, 259–261
  creating (KMDF example), 265–267
  creating (UMDF example), 261–264
  deleting, 116
  dispatch types (flow control), 56, 259, 385–386
    device scope and (synchronization), 388–389
  manual, retrieving requests from, 267–271
  in object hierarchy, 112–113
  power management for, 171, 196–197, 257–258
queue scope (synchronization), 387, 393–394

R

race conditions, 40, 41. *See also* synchronization
  canceled I/O requests, 56–57
raising IRQL, annotations for, 805
raw devices, 142–143
raw resource lists, 532
.rc files, 466, 622
READ_PORT_Xxx macros, 530
READ_REGISTER_Xxx macros, 530
read requests (I/O), 32, 225, 242
  event callbacks for, 282–291
  formatting for I/O targets, 331–334
  synchronous, 337
ReadFile function, 33, 225
real-time viewing of trace log, 436–437
recursive routines, 43
reference counts, 110, 113, 115, 117, 384
  COM object lifetime, 589–590, 609–610
  debugging, 591
  memory objects, 324, 443. *See also* memory objects
  tracking, 709–710
reference tracking, 689
references, circular, 113–114
refinement phase, SLAM engine, 831
reflector (UMDF), 76, 79
  fatal errors and, 80
REGISTER mapping. *See* memory mapping
registering callbacks, 54, 105. *See also* callbacks
registry access, 447–453
  KMDF registry objects and methods, 450–453
  UMDF device property store, 99, 447–450
registry key objects, 103, 114
  deleting, 116, 121
registry values for KMDF debugging, 713–714
release installations, 636, 656
  maintaining, 696
Release method (IUnknown), 113, 589, 590–591, 600, 609–610
ReleaseLock method (IWDFObject), 386
releases of WDK, 13
reliability, 61
remote I/O targets (KMDF), 309
  creating, 314–318
removal (device)
  power-down and, 184–187
  self-managed I/O during, 300–301
  surprise, 178
Remove method (IWDFIoTargetStateManagement), 319
/removedriver parameter (Driver Verifier), 681
reports, SDV, 842–847
ReqIsCancOnCancReq rule, 850
ReqIsNotCancelable rule, 850
ReqNotCanceled rule, 850
ReqNotCanceledLocal rule, 850
request buffer rules (KMDF), 850–851
request-cancellation rules (KMDF), 850
request-completion rules (KMDF), 849
RequestCompleted rule, 849
RequestCompletedLocal rule, 849
requesting callbacks at intervals. *See* timer objects
requests, I/O. *See* I/O processing
__reserved annotation, 761, 773
/reset parameter (Driver Verifier), 682
resource errors, detecting with PRE*f*ast, 733
resource files, 466, 622
resource identification, 531
resource list objects, 103, 114, 532–534
  deleting, 116
resource range list objects, 103, 114
  deleting, 116
resource requirements list objects, 103, 114
  deleting, 116
resources for more information, 20–22. *See also* documentation
restart (device), self-managed I/O during, 300
restoring IRQL, annotations for, 805
Results pane (SDV Defect Viewer), 845
Retrieve functions, 96, 104
RetrieveContext method, 125
RetrieveDeviceInstanceId method (IWDFDeviceInitialize), 150
RetrieveDevicePropertyStore method (IWDFDevice), 448

RetrieveDevicePropertyStore method (IWDFDeviceInitialize), 150, 448
RetrieveNextRequest method (IWDFIoQueue), 264, 268
RetrieveNextRequestByFileObject method (IWDFIoQueue), 264, 268
RetrieveUsbInterface method (IWDFUsbTargetDevice), 360, 363
RetrieveUsbPipeObject method (IWDFUsbInterface), 361, 364
return values for errors, 58
root folder, 14
root object (driver object), 110
routines for debugging, 699
rules, SDV, 826–831. *See also* SDV (Static Driver Verifier)
  hierarchies for, 854–855
  KMDF drivers, 847–855
  running verifications on, 839–840
  selecting, 834–839
RUN_WPP macro, 420–423, 628, 632
running drivers, debugging, 708–709

S

safe defaults, 62
sample drivers, obtaining, 16–17
saving IRQL, annotations for, 805
scatter/gather technique, 552, 554, 560–562
scheduling threads, 508–509. *See also* threads
scope, synchronization, 55, 387–398
  cancellation of I/O requests, 402–403
  KMDF drivers, 391–398
  SynchronizationScope attribute, 108, 391
  UMDF drivers, 390–391
SDDL (security descriptor definition language), 159–161
SDV Defect Viewer, 844–845
Sdv-map.h file, creating, 837–839, 856–857
SDV (Static Driver Verifier), 670, 823–862
  annotating KMDF code for, 832–834
  Fail_Driver3 sample driver (example), 855–859
  how it works, 825–831
  how to run, 834–842
  KMDF callback function role types for, 832–834, 860–862
  KMDF rules for, 847–855
  verification engine operation, 829–831
  viewing reports, 842–847
search phase, SLAM engine, 830–831
sections, INF files, 643
security, 61–62
  signing driver packages, 653–655
  trace messages and, 413, 417
  UMDF drivers, 86
security descriptors for device objects, 160–162
SecurityIdentification impersonation level, 276
SecurityImpersonation impersonation level, 276, 278
SECURITY_IMPERSONATION_LEVEL enumeration, 276
self-managed I/O, 297–305
  watchdog timer example (KMDF), 301–305
seminars on driver development, 22
send-and-forget flag (I/O completion), 338, 341, 343–345
Send method (IWDFIoRequest), 337, 343
  synchronous requests, 339–340, 369–370
sequential I/O request dispatching, 56, 259, 385
serialization, 41, 385
  automatic, 395–396, 539
    timer objects, 460
  synchronization scope and, 387–398
service routines, defined, 49
serviceability, 63–64
ServiceBinary directive, 648
services, drivers vs., 24
Set functions, 96, 104
SetBuffer method (IWDFMemory), 444
SetCompletionCallback method (IWDFIoRequest), 334
SetFilter method (IWDFDeviceInitialize), 150, 151, 192
SetLockingConstraint method (IWDFDeviceInitialize), 150, 151, 390
SetNamedValue method (IWDFNamedPropertyStore), 448, 450
SetPnpCapability method (IWDFDeviceInitialize), 150
SetPowerPolicyOwnership method (IWDFDeviceInitialize), 150, 151, 192
Setup action log, 664
Setup error log, 664
SetupAPI log, 664
SetupDiXxx functions, 146, 498–499
shared data, synchronized access to, 381–383
shutdown (device), 178–179
  device removal and, 184–187
  self-managed I/O during, 300–301
  shutdown sequences, 178
side-by-side operation of major versions, 63–64
side objects (UMDF), 79
signed catalog files, 643, 653–654
  specifying in INFs, 654
signing driver packages, 653–655
SimpleIO interface, 675–676
SimpleIOStress interface, 676
64-bit systems, as development requirement, 12
64-bit UMDF drivers, 85
Size attribute, 108
size of KMDF log, controlling, 728
Skeleton.rc file, 477, 488
Skeleton sample driver, 16, 475–493
SLAM verification engine, 829–831
slave devices. *See* system DMA devices

SLEEP scenario (PwrTest), 675
sleep states, 168
Sleep_Stress_With_IO.wsf script, 676
software-only drivers, Plug and Play and power management, 190–194
software tracing. *See* tracing
source code
  annotations in
    best practices, 816–818
    driver annotations, 774–778
    eliminating coding noise with, 751–752
    general-purpose, 760–774
    how to use, 752–760
    Osrusbfx2 sample driver (example), 818–822
    for SDV (Static Driver Verifier), 832–834
  noise in, 749–752, 815
  startup code, debugging, 706–708, 715
  trace message calls in, 431–432. *See also* tracing
Source Code pane (SDV Defect Viewer), 844
[SourceDisksFiles] section, INF files, 644
[SourceDisksNames] section, INF files, 644
Sources directory, cleaning, 836
Sources file, 621
  Fx2_Driver sample driver, 625–628
  modifying to run WPP preprocessor, 420–424
  Osrusbfx2 sample driver, 630–632
  Skeleton sample driver, 477
Sources file (Skeleton sample driver), 486–487
SOURCES macro, 627, 632
Special Pool option (Driver Verifier), 678, 680
specialized I/O targets, 309–310
  USB targets, 359–361
spin lock objects, 41, 44–45, 49, 103, 114, 399–401
  deleting, 116
  interrupt spin locks, 48, 400
  synchronized I/O request cancellation, 403–405
spin lock objects (WDFSPINLOCK), 523, 539
splitting I/O requests, 346–350
stability of UMDF drivers, 86
stack. *See* device stack; driver stack; kernel stack
STAMP directive (Makefile.inc), 629, 633
/standard parameter (Driver Verifier), 680
Start method (IWDFIoQueue), 260
Start method (IWDFIoTargetStateManagement), 319
Started state (I/O targets), 319, 340
startup (device), 178–179
  device enumeration and, 180–183
  self-managed I/O during, 300
startup code, debugging, 706–708, 715
state
  device state, 145
    system power state queries, 221
  I/O requests, to synchronize cancellation, 403–407
  I/O targets, 318–321, 340–341
state machines (KMDF), 72, 171
State pane (SDV Defect Viewer), 844–845
state variables, SDV rules for, 826
static code analysis tools, 669. *See also* PREfast for Drivers; SDV (Static Driver Verifier)
Static Driver Verifier (SDV), 670, 823–862
  annotating KMDF code for, 832–834
  Fail_Driver3 sample driver (example), 855–859
  how it works, 825–831
  how to run, 834–842
  KMDF callback function role types for, 832–834, 860–862
  KMDF rules for, 847–855
  verification engine operation, 829–831
  viewing reports, 842–847
static enumeration of child devices, 143
  in bus drivers, 157–158
statistic libraries for WDK, 14
STATUS_INVALID_DEVICE_REQUEST status, 503
STATUS_SUCCESS tests, 749
Stop method (IWDFIoQueue), 260
Stop method (IWDFIoTargetStateManagement), 320
Stopped for query-remove state (I/O targets), 319, 340
Stopped state (I/O targets), 319, 340
StopSynchronously method (IWDFIoQueue), 260
storage pass-through commands, 143
string annotations, 771–773
string objects, 103, 114
  deleting, 116, 120
[Strings] section, INF files, 644
STRSTR$ function, 783
success codes, 58–59
.SUFFIXES directive (Makefile.inc), 629
__super keyword, 609
surprise removal of device, 178
suspended I/O requests, 293, 296
symbol files, 19, 624, 702–704
symbolic links, 148, 160
SYNCH_LEVEL IRQLs, 512
synchronization, 31–32, 40–41, 49, 379–410. *See also* asynchronous I/O
  collection objects, 457
  I/O request cancellation (KMDF), 401–409
  I/O requests, 339–340
  processing at DIRQL, 547
  scope of, 55, 387–398
    cancellation of I/O requests, 402–403
    KMDF drivers, 391–398
    SynchronizationScope attribute, 108, 391
    UMDF drivers, 390–391
  spin lock objects, 41, 44–45, 49, 103, 114, 399–401
    deleting, 116
    interrupt spin locks, 48, 400
    synchronized I/O request cancellation, 403–405

synchronization (*continued*)
summary and general tips, 409–410
wait lock objects, 103, 114, 398–399
collection objects, 457
deleting, 116
WDF features, 384–386
WDF object hierarchy, 55
when required, 380–383
synchronization model (device property), 146
SynchronizationScope attribute, 108, 391
syntax misuses, 750
.sys files, 49, 623
system architecture (Windows), 25–27
system DMA devices, 550. *See also* DMA
System event log, 664
system memory, DMA abstraction and, 556. *See also* DMA
system power states, 168–169
system requirements for driver development, 12
system stability of UMDF drivers, 86
system thread context, 510

T

tagged memory allocation, 44
target objects, 103, 114, 147, 314
deleting, 116, 120, 121
TARGETLIBS macro, 627, 632
TARGETNAME macro, 626, 631
targets, I/O, 57, 98, 147, 307–321
creation and management, 313–321
deleting, 116
UMDF file handle targets, 353–354
UMDF implementation, 310–313
USB targets, 99, 355–376
TARGETTYPE macro, 626, 631
teardown, hardware, 531
template for UMDF drivers, 16, 475–493
test cases for annotations, 813–815
test installations, 636, 655–656, 711–714
test systems for drivers, 668–669
testing annotations, 813–814
testing WDF drivers, 667–696. *See also* debugging
best practices, 694–696
DMA drivers, 578–581
tools for, 63, 669, 695–696. *See also specific tool by name*
32-bit UMDF drivers, 85
threads, 39–41, 508–511
best practices, 527
blocking, 46
interrupt request levels. *See* IRQLs (interrupt request levels)
interruption scenarios, 518–523
IRQLs specific to, 512–515
priorities for, IRQLs vs., 38
work items and driver threads, 523–527
time-out values for I/O requests, 338–339
time value conversions, 461
timer objects, 104, 114, 459–464
deleting, 116, 120
synchronization scope, 395–396
Tlist.exe utility, 19, 699
TMF files, 434, 727
TMH file, including, 424
Toaster sample driver, 17
tools for driver development, 15
tools for driver testing, 63, 669, 695–696. *See also specific tool by name*
trace buffers, 414
trace consumers, 414–415
trace controllers, 413
trace logs, viewing, 435–437
trace message prefix, 729
trace preprocessor (Windows), 669
trace providers, 413, 425–427
trace sessions, 414, 433–439
Trace Tree pane (SDV Defect Viewer), 844
Tracefmt utility, 433, 437–439
Tracelog utility, 433, 437–439
Tracepdb utility, 433
Tracerpt utility, 433
TraceView utility, 433, 435–437
TraceWPP utility, 433
tracing, 411–439, 699
basics of, 412–417
best practices, 439
how to support, 420–432
message functions and macros, 417–419
running a trace session, 433–439
tools for, 63, 432–433
viewing messages with WinDbg, 725–726
tracking handles, 689
tracking reference counts, 709–710
training on driver development, 22
transfer direction, 355
transfer length (DMA devices), 554
transfer types (I/O), 34–35, 228–231
translated resource lists, 532
troubleshooting. *See* debugging
__try statements, annotating functions in, 790
typedef declarations, annotations on, 756–758

U

UMDF debugger extensions, 705
UMDF drivers, 67–70. *See also* WDF driver model
advantages and limitations, 86–88
applications and, 74
callback objects and interfaces, 54, 94, 96–97, 99–101. *See also* callbacks
callback implementation, 70, 585–586, 611–613

UMDF drivers (*continued*)
choosing between KMDF and, 89
COM as basis of. *See* COM (Component Object Model)
debugging, 87, 705. *See also* debugging
system requirements for, 12
development, system requirements for, 12
device object creation and initialization, 149–152
device property store, 99, 447–450
devices supported, 85
driver callback object creation, 135–137
errors in, 58, 80
how to build, 622–624
I/O. *See* I/O processing
installing, 635–666
basics of, 636–637
considerations, 637–642
INFs, creating, 643–648
INFs, examples of, 648–653
methods for, 655–663
package components, 642–643
signing and distributing packages, 653–655
troubleshooting, 663–666
object attributes, 94
power management
device enumeration and startup sequence, 181
power-down and removal sequence, 184
power-pageable devices, 174–176
power states, 169
protocol function driver (example), 197–201
software-only filter driver (example), 191–192
surprise-removal sequence, 188
sample driver. *See* Fx2_Driver sample driver
structure and requirements, 130–133, 584–586
template for, 16, 475–493
typical I/O requests to, 80–81
WDF debugger extension, 63
Windows API with, 497–500
UMDF host process, 78
UMDF infrastructure, 76–81
UMDF object model, 75, 95–102
context areas, 122–125
object creation, 106–107
object hierarchy, 112–114
object lifetime, 115–119
UMDF programming model, 68–70
UMDF stack, 78
UMDF Verifier, 690–693
UmdfDispatcher directive, 647
UmdfImpersonationLevel directive, 276, 647
UmdfLibraryVersion directive, 648
UmdfMethodNeitherAction directive, 230–232
UMDF_MINOR_VERSION macro, 626
UmdfService directive, 647
UmdfServiceOrder directive, 647
UMDFSkeleton_OSR.inx file, 477, 489–493
UMDFSkeleton_Root.inx file, 477, 489
UMDF_VERSION macro, 626
unbalanced I/O requests, 273
uncancelable I/O requests, 294
uninitialized variables, detecting, 741–742
uninstalling drivers, 659
unmapping hardware resources, 537
UnmarkCancelable (IWDFIoRequest), 294
UNREFERENCE_PARAMETER macro, 46
unsupported I/O requests, handling, 503–506
Up device object (UMDF device stack), 79, 81
updating drives with Device Manager, 659–663
upper edge, WDF frameworks, 74–75
USB device objects (WDFUSBDEVICE), 89, 104, 114, 147, 355–356, 359, 360. *See also* USB target objects
continuous reader for (KMDF), 375–376
data transfer models, 357–358
deleting, 116
in object hierarchy, 112
USB dispatchers, 312
USB I/O request completion parameters objects, 99
USB interface objects (WDFUSBINTERFACE), 99, 104, 114, 147, 359
deleting, 116
in object hierarchy, 112
USB pipe objects, 104, 114, 147, 359, 361
configuring, 363–364
deleting, 116
in object hierarchy, 112
USB target objects, 99, 355–376
configuring, 361–369
request completion, 99
sending I/O requests to, 369–375
USBD_PIPE_TYPE enumeration, 364
USE_MSVCRT macro, 628
user interfaces, drivers and, 87
user-mode debugging, 698
user-mode framework. *See* UMDF drivers
user-mode programming, 26
debugging, 87. *See also* debugging
issuing I/O requests, 32
memory access, 45. *See also* memory objects
threads, 508. *See also* threads
user mode (Windows system), 26, 49
UuidGen tool, 425–427
__uuidof utility, 610

V

validating
I/O control codes, 227
parameters, 62, 242
validation phase, SLAM engine, 830–831
VerboseOn (KMDF log controls), 729

!verifier debugger command, 523, 679
tracking fault and pool allocation counters, 686
using information during debugging, 684–686
Verifier.exe. *See* Driver Verifier
verifying drivers, 669. *See also* testing WDF drivers
Application Verifier, 693–694
DMA drivers, 578–580
Driver Verifier for, 46, 677–686
deadlock detection, 45, 678, 680
examples, 679–684
Force IRQL Checking option, 45, 678, 680, 683–684
IRQL problems, testing for, 523
testing DMA drivers, 578–579
KMDF Verifier, 45, 687–690
SDV for. *See* SDV (Static Driver Verifier)
UMDF Verifier, 690–693
VERIFY_IS_IRQL_PASSIVE_LEVEL( ) macro, 46
version resource file (Skeleton sample driver), 477, 488
[Version] section, INF files, 644, 654
versioning, 63–64, 637
versions of WDK, 13
versions of Windows, selecting, 619–620
virtual address space, 42–44
virtual function tables. *See* VTables
Vista. *See entries at* Windows Vista
vocabulary for driver development, 47–49
/volatile parameter (Driver Verifier), 681
VTables, 592–593

W

wait lock objects, 103, 114, 398–399
collection objects, 457
deleting, 116
wake support for KMDF drivers, 174, 207, 212–217, 220–222
example, 217–220
watch construct (SDV), 826
watchdog timer example (KMDF), 301–305
WDF architecture, 73–76
WDF bloggers, 21
WDF driver model, 51–64. *See also* KMDF drivers; UMDF drivers
defined, 52–53
I/O model, 55–59
interaction with Windows, 68
object model, 53–55
Plug and Play and power management, 59–61, 170–176, 189–190
defaults, 170
security, 61–62
serviceability and versioning, 63–64
synchronization requirements, 383
testing and tracing, 63
WDM drivers vs., 25, 51–52
!wdf extensions. *See* KMDF debugger extensions
WDF frameworks, 67–89. *See also* KMDF drivers; UMDF drivers
device and driver support, 84–89
distributions of, 638–639
object model, 72, 75
overview of KMDF, 71–73, 81–84
overview of UMDF, 68–70, 76–81
WDF memory objects. *See* memory objects
WDF object model, 91–128. *See also* KMDF object model; UMDF object model
attributes, 94, 108
context areas, 122–128
implementation, 95–105
object creation, 106–110, 149–157
object hierarchy and lifetime, 110–122. *See also* hierarchy of WDF objects
WDF support objects, 441–473
collection objects, 102, 114, 455–459
deleting, 115
general objects (base objects), 97, 454–455
deleting, 116
in object hierarchy, 112
memory allocation. *See* allocating memory
registry access, 447–453
KMDF registry objects and methods, 450–453
UMDF device property store, 99, 447–450
timer objects, 104, 114, 459–464
deleting, 116, 120
synchronization scope, 395–396
WMI support in KMDF drivers, 464–473
WDF update logs, 664
WDF verifiers, 63
WDF_ABS_TIMEOUT_IN_Xxx conversion functions, 461
WDF_CALLBACK_CONSTRAINT enumeration, 390
WDF_COMMON_BUFFER_CONFIG structure, 566
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME macro, 127, 154
WDF_DEVICE_IO_TYPE enumeration, 229–230
WDF_DEVICE_PNP_CAPABILITIES structure, 204
WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS structure, 208–210, 216
WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT macrostructure, 208
WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS structure, 214, 215–216
WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS_INIT macro, 216
WDF_DEVICE_POWER_STATE enumeration, 175
WDF_DMA_ENABLER_CONFIG structure, 566, 568
WDF_DRIVER_CONFIG_INIT function, 139
WDF_FILEOBJECT_CONFIG structure, 273
WDF_INTERRUPT_CONFIG structure, 539
WDF_IO_QUEUE_CONFIG structure, 194, 197, 204

WDF_IO_TARGET_OPEN_PARAMS structure, 314–316
WDF_IO_TARGET_OPEN_PARAMS_INIT_XXX functions, 314–315
WDF_NO_ATTRIBUTES structure, 539
WDF_OBJECT_ATTRIBUTES structure, 108, 139
WDF_OBJECT_ATTRIBUTES_INIT function, 108
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE macro, 127–128, 469
WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE macro, 127–128, 154
WDF_*Object*_CONFIG structures, 107
WDF_*Object*_CONFIG_INIT functions, 107
WDF_POWER_DEVICE_STATE enumeration, 206
WDF_POWER_POLICY_EVENT_CALLBACKS structure, 214, 216
WDF_POWER_POLICY_EVENT_CALLBACKS_INIT macro, 214, 217
WDF_PROPERTY_SDTORE_DISPOSITION enumeration, 448
WDF_REQUEST_PARAMETERS structure, 270, 291
WDF_REQUEST_SEND_OPTION_Xxx flags, 338
  WDF_REQUEST_SEND_OPTION_IGNORE_TARGET_STATE, 338, 340
  WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET, 338, 341, 343–345
WDF_SYNCHRONIZATION_SCOPE enumeration, 391
WDF_TIMER_CONFIG_INIT function, 460
WDF_TRI_STATE enumeration, 273
WDF_USB_DEVICE_SELECT_CONFIG_PARAMS structure, 365–367
WDF_USB_REQUEST_COMPLETION_PARAMS structure, 375
WDF_WMI_INSTANCE_CONFIG structure, 467, 469
WDF_WMI_PROVIDER_CONFIG structure, 466, 469
WDFCHILDLIST objects. *See* child list objects
WdfChildListXxx methods, 158
WDFCMRESLIST objects. *See* resource list objects
WdfCmResourceListGetCount method, 532
WdfCmResourceListGetDescriptor list, 532, 536
WDFCOLLECTION objects. *See* collection objects
WdfCollectionXxx methods, 456
  WdfCollectionAdd method, 408
WDFCOMMONBUFFER objects. *See* DMA common buffer objects
WdfCommonBufferCreate method, 566, 568–569
WdfCommonBufferCreateWithConfig method, 566, 568–569
WdfCommonBufferGetAlignedLogicalAddress method, 567, 569
WdfCommonBufferGetAlignedVirtualAddress method, 567, 569
WdfControlFinishInitializing method, 144
!wdfcrashdump debugger extension, 728
WDFDEVICE objects. *See* device objects
WdfDeviceAllocAndQueryProperty method, 450
WdfDeviceAssignMofResourceName method, 469
WdfDeviceAssignS0IdleSettings method, 209–212, 217, 218, 220, 853
WdfDeviceAssignSxWakeSettings method, 214, 215, 216, 853
WdfDeviceConfigureRequestDispatching method, 265, 267, 279
WdfDeviceCreate method, 155
WdfDeviceEnqueueRequest method, 252
WdfDeviceGetDevicePowerState method, 175
WdfDeviceGetIoTarget method, 313, 344
WDFDEVICE_INIT structure, 153, 155, 193
WdfDeviceInitialize method, 155
WdfDeviceInitXxx methods, 146, 153
  WdfDeviceInitAssignName, 160
  WdfDeviceInitAssignSDDLString, 160, 162
  WdfDeviceInitAssignWdfmIrpPreprocessCallback, 504–506
  WdfDeviceInitSetIoInCallerContextCallback, 250
  WdfDeviceInitSetIoType method, 229–230
  WdfDeviceInitSetPnpPowerEventCallbacks, 204, 215, 298–299
  WdfDeviceInitSetPowerNotPageable, 175
  WdfDeviceInitSetPowerPageable, 175
  WdfDeviceInitSetPowerPolicyEventCallback, 153, 217, 853
  WdfDeviceInitSetPowerPolicyOwnership, 143
WdfDeviceOpenRegistryKey method, 450–451
WdfDeviceQueryProperty method, 450
WdfDeviceResumeIdle method, 210
WdfDeviceSetAlignmentRequirement method, 568
WdfDeviceSetDeviceState method, 187
WdfDeviceSetSpecialFileSupport method, 174
WdfDeviceStopIdle method, 210–211, 220, 258
WdfDeviceWdmXxx methods, 501
WdfDevStateIsNP method, 175–176
WdfDmaDirectionReadFromDevice method, 571
WdfDmaDirectionWriteToDevice method, 571
WDFDMAENABLER objects. *See* DMA enabler objects
WdfDmaProfileXxx methods, 566, 568
WDFDMATRANSACTION objects. *See* DMA transaction objects
WdfDmaTransactionCreate method, 567, 569
WdfDmaTransactionDmaCompleted method, 572, 576–577
WdfDmaTransactionDmaCompletedFinal method, 575
WdfDmaTransactionDmaCompletedWithLength method, 576
WdfDmaTransactionExecute method, 570
WdfDmaTransactionGetBytesTransferred method, 573, 575, 577
WdfDmaTransactionInitialize method, 569, 571

WdfDmaTransactionInitializeUsingRequest method, 569, 571
WdfDmaTransactionRelease method, 572, 577
WDFDPC objects. *See* DPCs (deferred procedure calls)
WdfDpcWdmGetDpc method, 501
WDFDRIVER objects, 103, 114, 137
  deleting, 116
WdfDriverCreate method, 139
WdfExecutionLevelXxx execution levels, 396–398
WdfFdoInitWdmGetPhysicalDevice method, 501
WdfFdoInitXxx functions, 154
  registry objects and methods, 450–453
  WdfFdoInitSetDefaultChildListConfig, 157–158
  WdfFdoInitSetEventCallbacks, 153
  WdfFdoInitSetFilter, 154, 193
WDFFILEOBJECT objects. *See* file objects
WdfFileObjectWdmGetFileObject method, 501
WdfGetDriver method, 139
WdfGetRequestParameters method, 291
WDFINTERRUPT objects. *See* interrupt objects
WdfInterruptAcquireLock method, 515, 547
WdfInterruptCreate method, 540
WdfInterruptDisable method, 541
WdfInterruptEnable method, 541
WdfInterruptGetDevice method, 544–545
WdfInterruptQueueDpcForIsr method, 545
WdfInterruptReleaseLock method, 547
WdfInterruptSynchronize method, 515, 546–547
WdfInterruptWdmGetInterrupt method, 501
WdfIoQueueXxx methods, 260, 265–266, 268, 270–271
  WdfIoQueueRetrieveXxx methods, 252, 268, 270–271
WDFIORESLIST objects. *See* resource range list objects
WDFIORESREQLIST objects. *See* resource requirements list objects
WDFIOTARGET objects. *See* target objects
WdfIoTargetClose method, 319
WdfIoTargetCloseForQueryRemove method, 320, 321
WdfIoTargetCreate method, 314
WdfIoTargetFormatRequestXxx methods, 332–334, 338
WdfIoTargetGetState method, 319
WdfIoTargetOpen method, 314, 319
WdfIoTargetSendXxxSynchronously methods, 337–338
WdfIoTargetStart method, 319, 360
WdfIoTargetStop method, 320, 360
WdfIoTargetWdmXxx methods, 501
!wdfkd debugger extensions, 705
  !wdfkd.wdfdriverinfo, 689–690
WdfKd.dll library, 14
WDFKEY objects. *See* registry key objects
!wdflogdump debugger extension, 727
WDFLOOKASIDE objects. *See* lookaside list objects
WdfLookasideListCreate method, 445
WDFMEMORY objects. *See* memory objects
WdfMemoryXxx methods, 248, 444
  WdfMemoryAssignBuffer, 350
  WdfMemoryCreate, 327, 330, 444
  WdfMemoryCreateFromLookaside, 444, 446
  WdfMemoryCreatePreallocated, 327, 444
  WdfMemoryGetBuffer, 444, 446
WDFOBJECT objects. *See* generic objects
WdfObjectAcquireLock method, 386
WdfObjectAllocateContext method, 126, 128
WdfObjectCreate method, 109, 455
WdfObjectDelete method, 115, 120
  deleting memory objects, 447
WdfObjectDereference method, 115
WdfObjectReference method, 115
WdfObjectReleaseLock method, 386
WdfPdoInitAllocate method, 158
WdfPdoInitSetEventCallbacks method, 159
WdfPowerDeviceXxx constants, 206–207
WdfPropertyStoreCreateIfMissing method, 449
WDFQUEUE objects. *See* queue objects
WdfRegistryXxx methods, 450–453
  WdfRegistryOpenKey, 450
  WdfRegistryWdmGetHandle, 501
WDFREQUEST objects, 103, 114, 323
  deleting, 116
WdfRequestCancelSentRequest method, 352
WdfRequestComplete method, 292, 509
WdfRequestCompleteWithInformation method, 292
WdfRequestCompleteWithPriorityBoost method, 292, 509
WdfRequestCreate method, 323
WdfRequestCreateFromIrp method, 348–350, 501
WdfRequestFormatRequestUsingCurrentType method, 331
WdfRequestGetCompletionParams request, 336
WdfRequestGetStatus method, 339
WdfRequestIsCanceled method, 295
WdfRequestMarkCancelable method, 294
WdfRequestProbeAndLockUserBufferXxx, 252
WdfRequestRetrieveXxx methods, 247–251, 501, 570
  WdfRequestRetrieveInputMemory, 347
  WdfRequestRetrieveOutputMemory, 286, 347
WdfRequestReuse method, 347–350
WdfRequestSend method, 337, 344, 345
WdfRequestSetCompletionRoutine method, 334
WdfRequestStopActionPurge method, 352
WdfRequestStopActionSuspend method, 352
WdfRequestUnmarkCancelable method, 294, 403
WdfRequestWdmGetIrp method, 502
!wdfsettraceprefix debugger extension, 729
WDFSPINLOCK objects. *See* spin lock objects

WdfSpinLockCreate method, 539
WDFSTRING objects. *See* string objects
WDFTIMER objects. *See* timer objects
WdfTimerCreate method, 459–460, 463
WdfTimerGetParentObject method, 460
WdfTimerStart method, 460, 461, 463
WdfTimerStop method, 460, 463
!wdftmffile debugger extension, 727
WDFUSBDEVICE objects. *See* USB device objects
WDFUSBINTERFACE objects. *See* USB interface objects
WdfUsbInterfaceXxx methods, 359, 361, 368
WDFUSBPIPE objects. *See* USB pipe objects
WdfUsbTargetDeviceXxx methods, 359, 360, 365–367, 371–372
WdfUsbTargetPipeXxx methods, 359, 368, 372, 374, 502
  WdfUsbTargetPipeConfigContinuousReader, 375
WdfVerifierDbgBreakPoint method, 688
WDFVERIFY macro, 688
WDFWAITLOCK objects. *See* wait lock objects
WdfWaitLockAcquire method, collection objects, 457
WdfWaitLockCreate method, collection objects, 457
WdfWaitLockRelease method, collection objects, 457
WdfWdmDeviceGetWdfDeviceHandle method, 502
WdfWdmDriverGetWdfDriverHandle method, 502
WDFWMIINSTANCE objects. *See* WMI instance objects
WdfWmiInstanceCreate method, 469
WdfWmiInstanceRegister method, 467
WDFWMIPROVIDER objects. *See* WMI provider objects
WdfWmiProviderCreate method, 467
WDFWORKITEM objects. *See* work item objects
WdfWorkItemCreate method, 526
WdfWorkItemEnqueue method, 524, 526
  thread context, 510
WDK (Windows Driver Kit), obtaining, 13
WDK documentation, 14–15
WDK installation, 11–14
WDK root folder, 14
%wdk% root folder, 14
WDM drivers, 88, 500–503. *See also* kernel-mode programming
  WDF driver model vs., 25, 51–52
WDT_TIMER_CONFIG_INIT_PERIODIC function, 460
WDTF (Windows Device Testing Framework), 675–676
!wdtlogsave debugger extension, 728
WER (Windows Error Reporting), 692–693, 696
WHDC (Windows Hardware Developer Central), 20
WIN32_WINNT_VERSION macro, 628
WinDbg debugger, 18, 63, 698, 699–706
  commands, 702
  extensions, 704–706
  installation error debugging, 663
  KMDF debugging session, starting, 714–715
  suggestions for experimenting, 729–730
  viewing KMDF log, 726–729
  viewing trace messages, 725–726
WINDBG list server, 21
Windows API, 26, 86–87, 497–500
Windows Debugger Users List, 21
Windows Device Manager, 637
Windows Device Testing Framework, 675–676
Windows DMA abstraction, 555–563
Windows driver architecture, 27–28
Windows Error Reporting (WER), 692–693, 696
Windows File System Developers List, 21
Windows I/O model, 31–36, 239
Windows kernel, about, 24. *See also entries at* kernel
Windows logo, obtaining, 653
Windows symbols, obtaining, 19
Windows system architecture, 25–27
Windows System Software Developers List, 21
Windows trace preprocessor, 669
Windows version, selecting, 619–620
Windows Vista, elevated privileges for, 15
Windows Vista, ETW in, 416–417
Windows Vista, kernel debugging for, 711–712
WinHEC event, 22
with guard construct (SDV), 826
WMI handler (I/O requests), 240
WMI instance objects, 104, 114, 147, 467
  deleting, 116
  event callbacks, 470–473
WMI provider objects, 104, 114, 147, 466–467
  deleting, 116
WMI request handlers, 464
WMI support in KMDF drivers, 464–473
work item, defined, 49
work item objects, 104, 114, 523
  deleting, 116, 120
  driver threads and, 523–527
  synchronization scope, 395–396
WPP_CLEANUP macro, 429–431
WPP_CONTROL_GUIDS macro, 425–426
WPP_DEFINE_CONTROL_GUID macro, 425
WPP_INIT_TRACING macro, 427–429
WPP_LEVEL_ENABLED macro, 422
WPP_LEVEL_LOGGER macro, 422
WPP preprocessor, running, 420–424
WPP software tracing. *See* tracing
write requests (I/O), 32, 225, 242
  event callbacks for, 282–291
  formatting for I/O targets, 332–333
  synchronous, 337
WRITE_PORT_Xxx macros, 530
WRITE_REGISTER_Xxx macros, 530
WriteFile function, 33, 225
Wudfddi.idl file, 595
!wudfext debugger extensions, 705
WudfExt.dll library, 14
Wudfhost process, 78

Penny Orwick

Penny Orwick is an independent writer working with Steyer Associates, specializing in driver development and Windows operating system topics. She began writing about Windows drivers in 1997 and has worked with the Windows Driver Foundation team from the outset of the project to develop technical papers about WDF for the driver development community. Penny has a B.A. from Cornell University and an M.F.A. from the University of Montana.

Guy Smith

Guy Smith started programming in Fortran IV on punch cards as a Geophysics graduate student. He began writing SDK documentation for Microsoft early in 1996 and has since worked on many Microsoft technologies including Windows CE, the Windows Shell and Common Controls, DirectX 8, Internet Explorer, and the Windows Presentation Foundation. He now works as an independent writer with Steyer Associates, focusing on device drivers and related kernel-mode topics. Guy is a dedicated amateur early musician, performing the music of the Renaissance on lute, cittern, and serpent.

Carol Buchmiller

Carol Buchmiller has been writing and editing documentation for personal computer software since 1981. After working at Microsoft in the early 1980s, Carol has written for various software companies in the Pacific Northwest, eventually specializing in Windows kernel-mode drivers and hardware compatibility topics. In 2000, Carol joined Microsoft to write for the Windows Hardware Developer Central Web site. Carol has a B.A. from Whitman College and has completed certificate programs in C and C++.

Annie Pearson

Annie Pearson began writing and editing software documentation in 1982. She began writing and managing technical documentation projects for Windows with the Windows 3.1 Resource Kit. Annie has been the lead writer and information architect for the Windows Hardware Developer Central Web site since 1997.

Microsoft Press Support Information

Every effort has been made to ensure the accuracy of the contents of the book. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article. Microsoft Press provides support for books and companion CDs at the following Web site:

http://www.microsoft.com/learning/support/books/.

If you have comments, questions, or ideas regarding the book, or questions that are not answered by visiting the site above, please send them via e-mail to mspinput@microsoft.com.

In the United States, Microsoft software product support issues not covered by the Microsoft Knowledge Base are addressed by Microsoft Product Support Services. Location-specific software support options are available from

http://support.microsoft.com/gp/selfoverview/.

Microsoft Press provides corrections for books through the World Wide Web at

http://www.microsoft.com/mspress/support/.

To connect directly to the Microsoft Press Knowledge Base and enter a query regarding a question or issue that you may have, go to

http://www.microsoft.com/mspress/support/search.htm.

What do you think of this book?

We want to hear from you!

Do you have a few minutes to participate in a brief online survey?

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you.

To participate in our survey, please visit:

www.microsoft.com/learning/booksurvey/

...and enter this book's ISBN-10 number (appears above barcode on back cover*). As a thank-you to survey participants in the United States and Canada, each month we'll randomly select five respondents to win one of five $100 gift certificates from a leading online merchant. At the conclusion of the survey, you can enter the drawing by providing your e-mail address, which will be used for prize notification only.

Thanks in advance for your input. Your opinion counts!

*** Where to find the ISBN-10 on back cover**

ISBN-13: 000-0-0000-0000-0
ISBN-10: 0-0000-0000-0

Example only. Each book has unique ISBN.

Microsoft Press

No purchase necessary. Void where prohibited. Open only to residents of the 50 United States (includes District of Columbia) and Canada (void in Quebec). For official rules and entry dates see:

www.microsoft.com/learning/booksurvey/