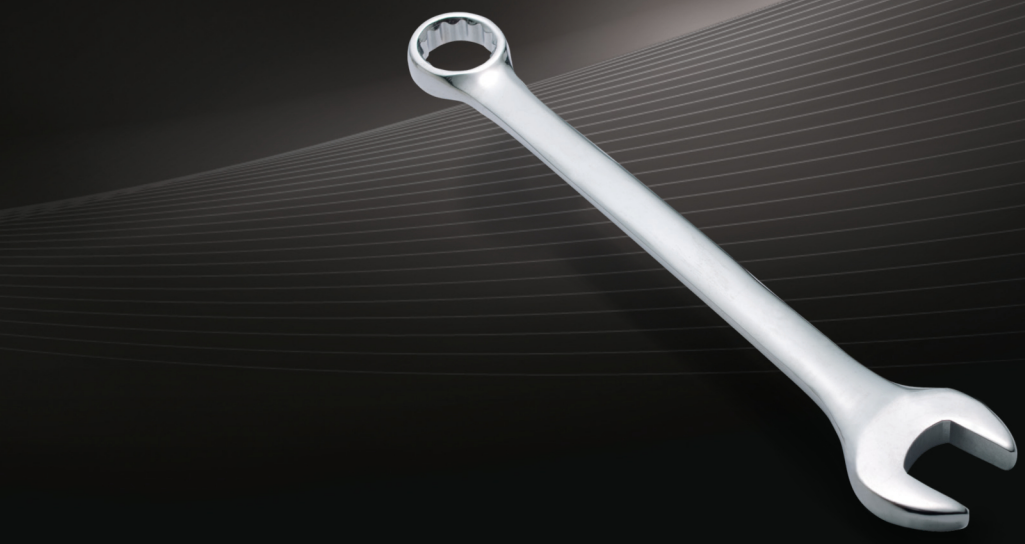
Microsoft

Inside Windows® Debugging

Practical Debugging and Tracing Strategies



Tarik Soulami

Inside Windows® Debugging:
Practical Debugging and Tracing Strategies

Use Windows debuggers throughout the development cycle—and build better software

Rethink your use of Windows debugging and tracing tools—and learn how to make them a key part of test-driven software development. Led by a member of the Windows Fundamentals Team at Microsoft, you'll apply expert debugging and tracing techniques—and sharpen your C++ and C# code analysis skills—through practical examples and common scenarios. Learn why experienced developers use debuggers in every step of the development process, and not just when bugs appear.

Discover how to:

- Go behind the scenes to examine how powerful Windows debuggers work
- Catch bugs early in the development cycle with static and runtime analysis tools
- Gain practical strategies to tackle the most common code defects
- Apply expert tricks to handle user-mode and kernel-mode debugging tasks
- Implement postmortem techniques such as JIT and dump debugging
- Debug the concurrency and security aspects of your software
- Use debuggers to analyze interactions between your code and the operating system
- Analyze software behavior with the Event Tracing for Windows (ETW) framework

Get code samples on the web
Ready to download at
http://go.microsoft.com/FWLink/?Linkid=245713

*For **system requirements**, see the Introduction.*

ISBN: 978-0-7356-6278-0

9 780735 662780 90000

microsoft.com/mspress

U.S.A. $39.99
Canada $41.99
[*Recommended*]

Programming/Windows Debugging

About the Author

Tarik Soulami, a principal development lead on the Windows Fundamentals Team, has more than 10 years of experience designing and developing system-level software at Microsoft. Before joining the Windows team, Tarik spent several years on the Common Language Runtime (CLR) Team, where he helped shape early releases of the Microsoft® .NET framework.

Microsoft®

Inside Windows® Debugging

Tarik Soulami

ISBN: 978-0-7356-6278-0

2 3 4 5 6 7 8 9 10 LSI 8 7 6 5 4 3

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.

Acquisitions Editor: Russell Jones

Developmental Editor: Russell Jones

Production Editor: Melanie Yarbrough

Editorial Production: Waypoint Press

Technical Reviewer: John Mueller

Copyeditor: Roger LeBlanc

Indexer: Christina Yeager

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

Illustrator: Steve Sagman

[2013-05-17]

Contents at a Glance

Contents

Foreword

Like many others, I am a firm believer in using tools to expand our understanding of how systems really work. In fact, I began my career as a performance tools developer. My boss at that time had many simple sayings; among them was one of my favorites, "Our team is only as good as our people and our people are only as good as our tools." As a manager, I've made it a priority to ensure we dedicate many of our top engineers to tools development and I have always encouraged using tools to teach and grow engineers.

Teaching books such as Tarik's are meant to help improve the productivity, understanding, and confidence of others. Being an individual who has had both a lifelong passion for tools and learning, as well as someone who has spent more than a decade and a half working in Windows, it is an honor and a pleasure for me to write the foreword to Tarik's insightful book.

For decades, the Windows team has worked tirelessly to improve the core capabilities of the platform and to make it more suitable for increasingly diverse hardware configurations and software stacks. This hard work has paid off; Windows today is the preeminent platform for developers, consumers, and businesses across the globe. With more than 1 billion PCs and customers, Windows is both the market-leading server and client computing platform. Variants of Windows run on small form-factor mobile devices, within embedded intelligent systems, on uniform and non-uniform (NUMA) memory architectures, on the XBOX gaming console, and on single CPU systems as well as those with hundreds of processors. Windows runs in clustered configurations with failover capabilities and sits atop hypervisors and Virtual Machines. And of course, Windows is in the cloud.

Enormous platform success and diversity can be accompanied by an equally enormous and diverse set of technical challenges. Thousands of engineers at Microsoft, and tens to hundreds of thousands outside of Microsoft, are involved in building, debugging, and troubleshooting a multitude of diverse configurations and solutions. To understand issues in a vastly diverse problem space, foundational tools, techniques, and concepts are essential.

To that end, Tarik has done an excellent job in detailing how a set of key foundational Windows tools work. In detailing these tools, Tarik succeeds in expanding the reader's awareness of key operating system concepts, architectures, strengths, and limitations. With the concepts understood and the tools mastered, readers can expect to be able to tackle all types of performance and debugging challenges. I believe Tarik's book will become a staple for a broad set of individuals, from novices to experts.

Michael Fortin, Ph.D.
Distinguished Engineer, Windows Fundamentals
Microsoft Corporation

Introduction

One exciting aspect of software programming is that there are usually many ways to accomplish the same goal. Unfortunately, this also presents software engineers with unique challenges when trying to make the best design or implementation choice for each situation. Experience plays a major role, and the learning process is often progressive as one learns to avoid past mistakes. Sadly, though, experience is often a variable concept. I have met several software engineers who, after spending a very long time working on an area, still lacked a basic understanding of how it *really* worked beyond the repetitive day-to-day tasks they grew accustomed to. I have also met others who have perfected their craft in a field after only a few years of working experience.

This book introduces a few techniques for methodically approaching software development problems primarily using the two "Swiss Army knives" of expert Microsoft Windows developers—namely, the Windows debuggers (WinDbg) and the Windows Performance Toolkit (Xperf). By focusing on *why* features and components work the way they do in the system rather than simply on *how* they work or *what* they do, this book tries to accelerate the process of learning by experience and to minimize the number of mistakes made when approaching new problems. An important part of the process is learning to compare and contrast with known solutions to existing ones.

Software engineering is still inherently a practical science. (Some might even argue it's an art rather than a science.) While there is certainly no substitute for real experience, the topic can definitely be approached with the same methodical persistence that works so well for other scientific disciplines. In fact, this approach works even better in software engineering because all behaviors can be explained rationally. After all, it is all just code—whether it's your own code or code written by others that you end up consuming in your software—and code can always be traced and understood.

Although this book deals with several architectural pillars of the Windows operating system as part of its debugging and tracing experiments, my main goal in writing it is less about covering those details and more about encouraging and developing this critical mindset. I hope to demonstrate how this approach can be used for solving a few interesting problems as part of this book, and that you continue to systematically apply debugging and tracing as you expand your learning beyond the topics directly covered here.

This book is not really about teaching native or managed code programming, either, although you'll find several good coding examples in the companion source code. Because it takes an inside-out look at how to explore the system using debugging

and tracing tools, this book will probably appeal more to software engineers with the desire to understand system internals rather than those with a need to quickly learn how to make use of a specific technology. However, I believe the approach and mindset it aspires to inculcate are applicable regardless of the technology or level of expertise. In fact, contrary to what many think, the higher the level of technology involved, the harder it becomes to grasp what goes on behind the scenes and the more expertise is needed in order to investigate failures when things inevitably go awry and require debugging skills to save the day. In pure C, for example, a call to *malloc* is just that: a function call. In C++, a call to the *new* keyword is emitted by the compiler as a call to the *new* operator function to allocate memory for the object, followed by code to construct the object (again, emitted by the compiler to possibly initialize a virtual pointer and invoke the constructors of the base classes, construct member data objects, and finally invoke the user-provided constructor code for the target leaf class). In C# (.NET), things get even more involved, because a one-line call to the *new* keyword might involve compiling new code at runtime, performing security checks by the .NET execution engine, loading the modules where the target type is defined, tracking the object reference for later garbage collection, and so on.

Who Should Read This Book

This book is aimed at software engineers who desire to take their game to the next level, so to speak, and perfect their mastery of Windows as a development platform through the use of debugging and tracing tools.

Assumptions

Readers should have basic familiarity with the C/C++ and C# programming languages. A basic knowledge of the Win32 and .NET platforms is helpful, but not strictly required, because this book makes every effort to introduce basic concepts before expanding into more advanced topics.

Organization of This Book

This book is divided into three parts:

- Part 1, "A Bit of Background," provides a brief overview of Windows development frameworks and the layers in the operating system that support them. This

basic knowledge is important to have when trying to make sense of the data surfaced by debugging and tracing tools.

- Part 2, "Debugging for Fun and Profit," covers the architectural foundations of debuggers in the Windows operating system. It also presents a number of extensible strategies that will help you make the most of the Windows debuggers. In addition, this part also shows how to use the WinDbg debugger to better understand system internals by analyzing the important interactions between your code and the operating system.
- Part 3, "Observing and Analyzing Software Behavior," continues this theme. It presents the Event Tracing for Windows (ETW) technology and illustrates how to leverage it in debugging and profiling investigations.
- Finally, you'll find two short appendices at the end of the book that recap the most common debugging tasks and how to accomplish them using WinDbg.

The table of contents will help you locate chapters and sections quickly. In addition, each chapter starts with a list of the main points covered in the chapter, and concludes with a summary section. You can also download the source code for all the experiments and examples shown throughout the book.

Conventions in This Book

This book presents information using conventions (listed in the following table) designed to make the information readable and easy to follow:

Convention	Meaning
Sidebars	Boxed sidebars feature additional bits of information that might be helpful for a given subject.
Notes	Notes provide useful observations related to the content discussed in the main text.
Inline Code	Inline code—that is, code that appears within a paragraph—is shown in *italic* font.
Code Blocks	Code blocks are shown in a different font to help you distinguish code from text easily. Important statements appear in bold font to help you focus on those aspects of the code.
Debugger Listings	Debugger listings are shown in a different font, and important commands are bolded to highlight them. The listings are also often prefixed with the output from the standard *vertarget* debugger command, which displays the OS version and CPU architecture where the experiment was conducted.
Function Names	Function names are sometimes referenced using their WinDbg symbolic names. For example, *kernel32!CreateFileW* refers to the CreateFileW Win32 API ("W" for the Unicode flavor) exported by the *kernel32.dll* module.

System Requirements

You will need the following hardware and software to follow the experiments and code samples in this book:

- **Operating System** Windows Vista or later. Windows 7 (or Windows Server 2008 R2) is highly recommended.
- **Hardware** Any computer that supports the Windows 7 operating system (OS) requirements. Except for the live kernel debugging experiments, a second computer to serve as a host kernel-mode debugger machine is typically required for kernel debugging.

Note The target and host don't really need to be separate physical machines. A common kernel debugging configuration—detailed in Chapter 2, "Getting Started"—is to run the target machine where you conduct the experiments as a Windows 7 virtual OS on a Windows Server 2008 R2 physical host computer, and run the kernel debugger in the host OS.

- **Hard disk** 1 GB of free hard-disk space to download and save the Windows Software Development Kit (SDK) and Driver Development Kit (DDK) ISO images. 40 MB of free hard-disk space to download and compile the companion source code. An additional 3 GB is required for installing Microsoft Visual Studio 2010.
- **Software** The following tools are used in the debugging and tracing examples shown throughout this book:
  - Version 7.1 of the Windows 7 SDK, which can be downloaded from the Microsoft Download Center at *http://www.microsoft.com/download/en/details.aspx?id=8442*. Both the Windows Debuggers (WinDbg) and Windows Performance Toolkit (Xperf) are part of this SDK.
  - The Application Verifier tool, which can also be downloaded from the Microsoft Download Center at *http://www.microsoft.com/download/en/details.aspx?id=20028*.
  - The System Internals suite of developer tools, which can be downloaded from *http://technet.microsoft.com/en-us/sysinternals/bb842062*.
  - Visual Studio 2010, any edition (excluding the free, stripped-down Express edition). The Visual Studio Ultimate edition is preferred because some advanced features, such as static code analysis and performance profil-

ing, are not supported in other editions. The 90-day, free trial version of Visual Studio offered by Microsoft, which can be downloaded at *http://www.microsoft.com/download/en/details.aspx?id=12187*, should suffice.

- The Windows 7 Driver Development Kit (DDK) is used to compile the companion source code and can also be downloaded from the Microsoft Download Center at *http://www.microsoft.com/download/en/details.aspx?id=11800*.

Code Samples

Most of the chapters in this book include experiments and examples that let you interactively try out new material introduced in the main text. The programs used in these experiments can be downloaded from the following page:

http://www.microsoftpressstore.com/title/9780735662780

Follow the instructions to download the Inside_Windows_Debugging_Samples.zip file.

Installing the Code Samples

Follow these steps to install the code samples on your computer so that you can use them with the experiments and examples provided in this book:

1. Unzip the Inside_Windows_Debugging_Samples.zip file that you downloaded from the book's website into a folder named *Book*\Code.

Warning Do not use directory names with spaces in the path hierarchy that you choose to download the samples to. The DDK build environment, which will be described shortly in the "Compiling the Code Samples" section, will fail to compile the source code if you do. It is recommended that you use *Book**Code* as the root of the source code because this is the assumed location used when referencing the programs in the main text. The samples in the companion source code are organized by chapter, so there should be a folder for every chapter under this root path location.

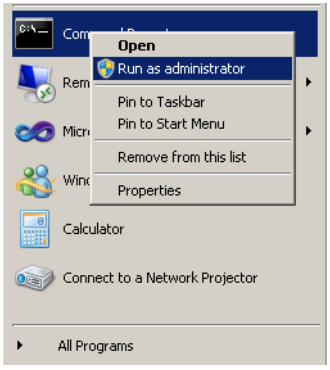
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the Inside_Windows_Debugging_Samples.zip file.

Running the Code Samples

The code samples are organized by chapter and are referenced in the book main text by their respective directory path locations to help you find them easily.

Local Administrator rights are required by some sample programs. In Windows Vista and later, a command prompt must be launched as elevated in order to have full administrative privileges even when the user account is a member of the local built-in Administrators security group. To do so in Windows 7, for example, you need to right-click the Command Prompt menu item in the Windows Start menu, and select Run As Administrator, as shown in the following screen shot:



Compiling the Code Samples

The supporting programs used in the experiments presented in the book main text fall into three categories:

- **C++ samples** The binaries for these programs are deliberately omitted from the downloadable ZIP file. When you compile these code samples locally, WinDbg locates their symbols and source code files automatically. So the experiments presented in the early chapters of this book will "just work" in this configuration, without needing to specify the source and symbol locations explicitly in WinDbg. The steps to compile all the native C++ code samples at once are included later in this section.

- **C# (.NET) samples** For convenience, the compiled .NET programs are included in the downloadable ZIP file. You can use them as-is and you won't lose much, given WinDbg doesn't support source-level .NET debugging, but you can also follow the instructions included in the following section to recompile them if you prefer.
- **JavaScript and Visual Basic samples** These scripts are interpreted by the corresponding scripting engines and do not require compilation.

Compiling the .NET Code Samples

Compiling the .NET code samples from the companion source code requires version 4.0 of the Microsoft .NET Framework or later. Though this version isn't installed by default on Windows 7, .NET Framework 4.0 gets installed by many other dependent programs, such as Visual Studio 2010. You can also download and install a standalone version from the Microsoft download center at *http://www.microsoft.com/download/en/details.aspx?id=17851*.

Each C# code sample from the companion source code will have a helper compilation script under the same directory. This script uses the .NET 4.0 C# compiler directly and is easy to invoke, as illustrated in the following command:

```
C:\book\code\chapter_04\LoadException>compile.bat
```

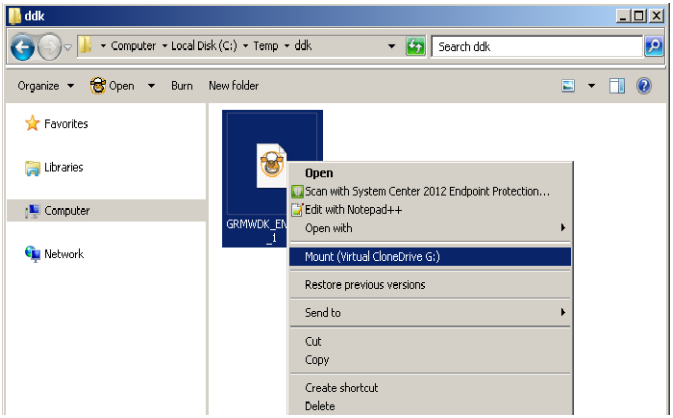
If this script fails to find the C# compiler, you should verify that you've installed .NET 4.0 in the default directory location where this script expects it to exist. If .NET 4.0 exists but was installed to a different location on your system, you need to modify the script and provide that path instead.

Compiling the C/C++ Code Samples

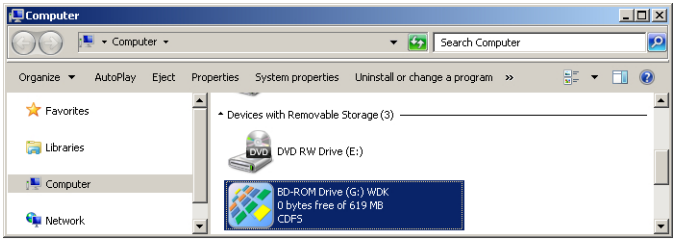
You can compile the C/C++ code samples from the companion source code using the Windows 7 Driver Development Kit (DDK) build tools. The following steps detail this process. I strongly recommend that you complete these steps before you start reading, because you'll need the code samples to follow the experiments shown later in this book.

1. Download the Windows 7 DDK ISO image from the Microsoft Download Center at *http://www.microsoft.com/download/en/details.aspx?id=11800*, and save it to your local hard drive. Set plenty of time aside for this download if you have a slow Internet connection; the DDK ISO file is over 600 MB in size.

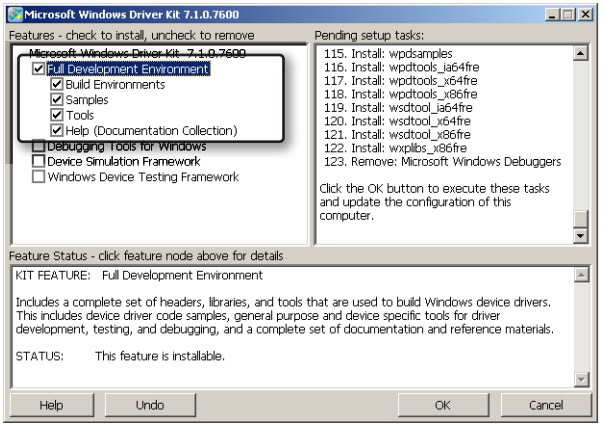
2. After the download is complete, mount the saved ISO file into a drive letter. There are several free tools for mounting ISO images on Windows. Virtual Clone Drive, which you can find on the Internet, is good freeware that works well both on Windows Vista and Windows 7. With that freeware installed, you should be able to right-click the ISO file and mount it, as shown in the following screen shot:



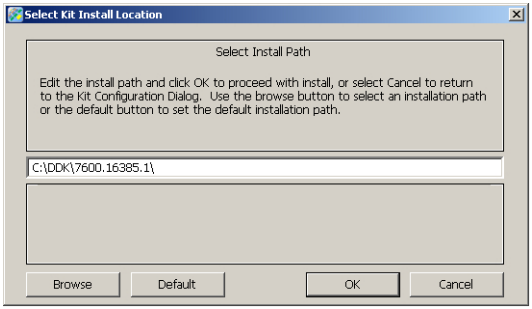
3. Double-click the newly mounted drive to kick off the DDK setup program, as shown here:



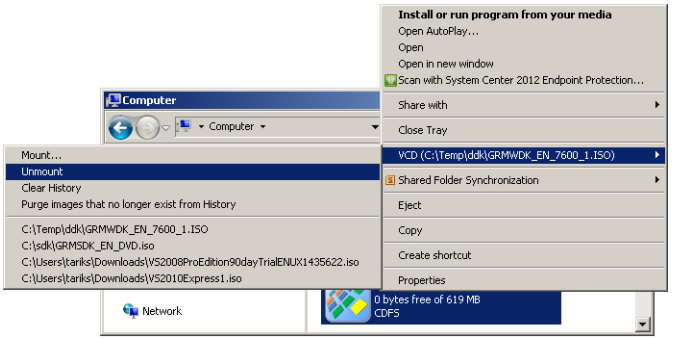
4. Select the Full Development Environment components from the DDK.



5. Then Install the components to the C:\DDK\7600.16385.1 directory, as shown in the following screen shot. This step will take several minutes to complete.



6. You can now unmount the DDK drive by right-clicking the mounted drive letter and selecting the Unmount menu action. This concludes this one-time installation of the Windows DDK build tools.



7. To build *x86* binaries, first start a command prompt window and type the following command:

```
C:\DDK\7600.16385.1>bin\setenv.bat c:\DDK\7600.16385.1
```

8. You can then build all the native code samples at once. Simply navigate to the root directory that you extracted the companion source code to and issue the following command. It should take only a minute or so to build all the C/C++ code from the companion source code:

```
C:\book\code>bcz
```

Acknowledgments

I am indebted to Vance Morrison for reviewing my rough drafts. Vance has been a role model for me since I joined Microsoft, and his critical insight was tremendously helpful in improving the quality of my writing. I only hope that I came close to his high standards.

I'd also like to thank Silviu Calinoiu and Shawn Farkas for their detailed review of my draft chapters. Their attention to detail and precise feedback was tremendously helpful.

Kalin Toshev provided valuable technical feedback during the formative stages of this book, and his unwavering dedication to test-driven software development really inspired me to write this book. This book owes a lot to him.

Ajay Bhave, Cristian Levcovici, and Rico Mariani provided several ideas for improving the material covered in this book. This book certainly wouldn't be the same without their help and guidance.

Special thanks to Michael Fortin for writing the foreword, and to all my colleagues in the Windows fundamentals team for their support during the writing of this book.

Finally, I have to acknowledge my family, who play an important part in my life. I would like to dedicate this book to my parents for providing me with the opportunity to reach for my dreams and always being there when I needed them. Your love and support mean the world to me.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735662780

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

Introduction

One exciting aspect of software programming is that there are usually many ways to accomplish the same goal. Unfortunately, this also presents software engineers with unique challenges when trying to make the best design or implementation choice for each situation. Experience plays a major role, and the learning process is often progressive as one learns to avoid past mistakes. Sadly, though, experience is often a variable concept. I have met several software engineers who, after spending a very long time working on an area, still lacked a basic understanding of how it *really* worked beyond the repetitive day-to-day tasks they grew accustomed to. I have also met others who have perfected their craft in a field after only a few years of working experience.

This book introduces a few techniques for methodically approaching software development problems primarily using the two "Swiss Army knives" of expert Microsoft Windows developers—namely, the Windows debuggers (WinDbg) and the Windows Performance Toolkit (Xperf). By focusing on *why* features and components work the way they do in the system rather than simply on *how* they work or *what* they do, this book tries to accelerate the process of learning by experience and to minimize the number of mistakes made when approaching new problems. An important part of the process is learning to compare and contrast with known solutions to existing ones.

Software engineering is still inherently a practical science. (Some might even argue it's an art rather than a science.) While there is certainly no substitute for real experience, the topic can definitely be approached with the same methodical persistence that works so well for other scientific disciplines. In fact, this approach works even better in software engineering because all behaviors can be explained rationally. After all, it is all just code—whether it's your own code or code written by others that you end up consuming in your software—and code can always be traced and understood.

Although this book deals with several architectural pillars of the Windows operating system as part of its debugging and tracing experiments, my main goal in writing it is less about covering those details and more about encouraging and developing this critical mindset. I hope to demonstrate how this approach can be used for solving a few interesting problems as part of this book, and that you continue to systematically apply debugging and tracing as you expand your learning beyond the topics directly covered here.

This book is not really about teaching native or managed code programming, either, although you'll find several good coding examples in the companion source code. Because it takes an inside-out look at how to explore the system using debugging

and tracing tools, this book will probably appeal more to software engineers with the desire to understand system internals rather than those with a need to quickly learn how to make use of a specific technology. However, I believe the approach and mindset it aspires to inculcate are applicable regardless of the technology or level of expertise. In fact, contrary to what many think, the higher the level of technology involved, the harder it becomes to grasp what goes on behind the scenes and the more expertise is needed in order to investigate failures when things inevitably go awry and require debugging skills to save the day. In pure C, for example, a call to *malloc* is just that: a function call. In C++, a call to the *new* keyword is emitted by the compiler as a call to the *new* operator function to allocate memory for the object, followed by code to construct the object (again, emitted by the compiler to possibly initialize a virtual pointer and invoke the constructors of the base classes, construct member data objects, and finally invoke the user-provided constructor code for the target leaf class). In C# (.NET), things get even more involved, because a one-line call to the *new* keyword might involve compiling new code at runtime, performing security checks by the .NET execution engine, loading the modules where the target type is defined, tracking the object reference for later garbage collection, and so on.

Who Should Read This Book

This book is aimed at software engineers who desire to take their game to the next level, so to speak, and perfect their mastery of Windows as a development platform through the use of debugging and tracing tools.

Assumptions

Readers should have basic familiarity with the C/C++ and C# programming languages. A basic knowledge of the Win32 and .NET platforms is helpful, but not strictly required, because this book makes every effort to introduce basic concepts before expanding into more advanced topics.

Organization of This Book

This book is divided into three parts:

- Part 1, "A Bit of Background," provides a brief overview of Windows development frameworks and the layers in the operating system that support them. This

basic knowledge is important to have when trying to make sense of the data surfaced by debugging and tracing tools.

- Part 2, "Debugging for Fun and Profit," covers the architectural foundations of debuggers in the Windows operating system. It also presents a number of extensible strategies that will help you make the most of the Windows debuggers. In addition, this part also shows how to use the WinDbg debugger to better understand system internals by analyzing the important interactions between your code and the operating system.
- Part 3, "Observing and Analyzing Software Behavior," continues this theme. It presents the Event Tracing for Windows (ETW) technology and illustrates how to leverage it in debugging and profiling investigations.
- Finally, you'll find two short appendices at the end of the book that recap the most common debugging tasks and how to accomplish them using WinDbg.

The table of contents will help you locate chapters and sections quickly. In addition, each chapter starts with a list of the main points covered in the chapter, and concludes with a summary section. You can also download the source code for all the experiments and examples shown throughout the book.

Conventions in This Book

This book presents information using conventions (listed in the following table) designed to make the information readable and easy to follow:

Convention	Meaning
Sidebars	Boxed sidebars feature additional bits of information that might be helpful for a given subject.
Notes	Notes provide useful observations related to the content discussed in the main text.
Inline Code	Inline code—that is, code that appears within a paragraph—is shown in *italic* font.
Code Blocks	Code blocks are shown in a different font to help you distinguish code from text easily. Important statements appear in bold font to help you focus on those aspects of the code.
Debugger Listings	Debugger listings are shown in a different font, and important commands are bolded to highlight them. The listings are also often prefixed with the output from the standard *vertarget* debugger command, which displays the OS version and CPU architecture where the experiment was conducted.
Function Names	Function names are sometimes referenced using their WinDbg symbolic names. For example, *kernel32!CreateFileW* refers to the CreateFileW Win32 API ("W" for the Unicode flavor) exported by the *kernel32.dll* module.

System Requirements

You will need the following hardware and software to follow the experiments and code samples in this book:

- **Operating System** Windows Vista or later. Windows 7 (or Windows Server 2008 R2) is highly recommended.
- **Hardware** Any computer that supports the Windows 7 operating system (OS) requirements. Except for the live kernel debugging experiments, a second computer to serve as a host kernel-mode debugger machine is typically required for kernel debugging.

Note The target and host don't really need to be separate physical machines. A common kernel debugging configuration—detailed in Chapter 2, "Getting Started"—is to run the target machine where you conduct the experiments as a Windows 7 virtual OS on a Windows Server 2008 R2 physical host computer, and run the kernel debugger in the host OS.

- **Hard disk** 1 GB of free hard-disk space to download and save the Windows Software Development Kit (SDK) and Driver Development Kit (DDK) ISO images. 40 MB of free hard-disk space to download and compile the companion source code. An additional 3 GB is required for installing Microsoft Visual Studio 2010.
- **Software** The following tools are used in the debugging and tracing examples shown throughout this book:
  - Version 7.1 of the Windows 7 SDK, which can be downloaded from the Microsoft Download Center at *http://www.microsoft.com/download/en/details.aspx?id=8442*. Both the Windows Debuggers (WinDbg) and Windows Performance Toolkit (Xperf) are part of this SDK.
  - The Application Verifier tool, which can also be downloaded from the Microsoft Download Center at *http://www.microsoft.com/download/en/details.aspx?id=20028*.
  - The System Internals suite of developer tools, which can be downloaded from *http://technet.microsoft.com/en-us/sysinternals/bb842062*.
  - Visual Studio 2010, any edition (excluding the free, stripped-down Express edition). The Visual Studio Ultimate edition is preferred because some advanced features, such as static code analysis and performance profil-

ing, are not supported in other editions. The 90-day, free trial version of Visual Studio offered by Microsoft, which can be downloaded at *http://www.microsoft.com/download/en/details.aspx?id=12187*, should suffice.

- The Windows 7 Driver Development Kit (DDK) is used to compile the companion source code and can also be downloaded from the Microsoft Download Center at *http://www.microsoft.com/download/en/details.aspx?id=11800*.

Code Samples

Most of the chapters in this book include experiments and examples that let you interactively try out new material introduced in the main text. The programs used in these experiments can be downloaded from the following page:

http://www.microsoftpressstore.com/title/9780735662780

Follow the instructions to download the Inside_Windows_Debugging_Samples.zip file.

Installing the Code Samples

Follow these steps to install the code samples on your computer so that you can use them with the experiments and examples provided in this book:

1. Unzip the Inside_Windows_Debugging_Samples.zip file that you downloaded from the book's website into a folder named *Book*\Code.

Warning Do not use directory names with spaces in the path hierarchy that you choose to download the samples to. The DDK build environment, which will be described shortly in the "Compiling the Code Samples" section, will fail to compile the source code if you do. It is recommended that you use *Book**Code* as the root of the source code because this is the assumed location used when referencing the programs in the main text. The samples in the companion source code are organized by chapter, so there should be a folder for every chapter under this root path location.

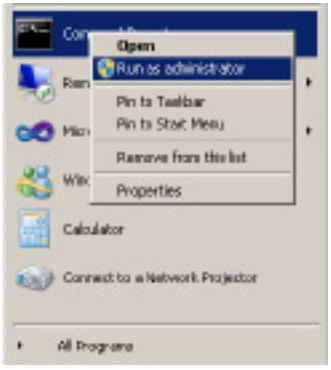
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the Inside_Windows_Debugging_Samples.zip file.

Running the Code Samples

The code samples are organized by chapter and are referenced in the book main text by their respective directory path locations to help you find them easily.

Local Administrator rights are required by some sample programs. In Windows Vista and later, a command prompt must be launched as elevated in order to have full administrative privileges even when the user account is a member of the local built-in Administrators security group. To do so in Windows 7, for example, you need to right-click the Command Prompt menu item in the Windows Start menu, and select Run As Administrator, as shown in the following screen shot:



Compiling the Code Samples

The supporting programs used in the experiments presented in the book main text fall into three categories:

- **C++ samples** The binaries for these programs are deliberately omitted from the downloadable ZIP file. When you compile these code samples locally, WinDbg locates their symbols and source code files automatically. So the experiments presented in the early chapters of this book will "just work" in this configuration, without needing to specify the source and symbol locations explicitly in WinDbg. The steps to compile all the native C++ code samples at once are included later in this section.

- **C# (.NET) samples** For convenience, the compiled .NET programs are included in the downloadable ZIP file. You can use them as-is and you won't lose much, given WinDbg doesn't support source-level .NET debugging, but you can also follow the instructions included in the following section to recompile them if you prefer.
- **JavaScript and Visual Basic samples** These scripts are interpreted by the corresponding scripting engines and do not require compilation.

Compiling the .NET Code Samples

Compiling the .NET code samples from the companion source code requires version 4.0 of the Microsoft .NET Framework or later. Though this version isn't installed by default on Windows 7, .NET Framework 4.0 gets installed by many other dependent programs, such as Visual Studio 2010. You can also download and install a standalone version from the Microsoft download center at *http://www.microsoft.com/download/en/details.aspx?id=17851*.

Each C# code sample from the companion source code will have a helper compilation script under the same directory. This script uses the .NET 4.0 C# compiler directly and is easy to invoke, as illustrated in the following command:

```
C:\book\code\chapter_04\LoadException>compile.bat
```

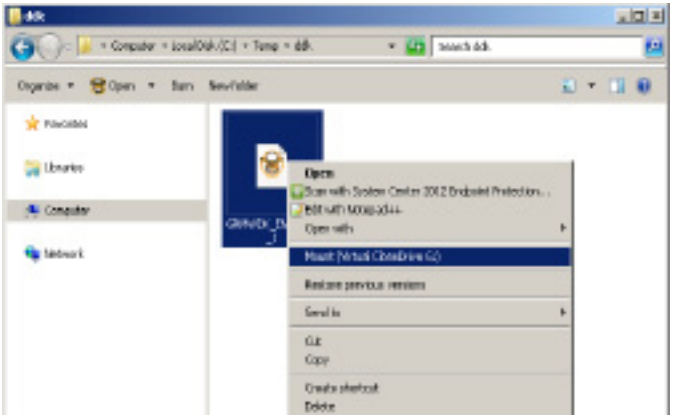
If this script fails to find the C# compiler, you should verify that you've installed .NET 4.0 in the default directory location where this script expects it to exist. If .NET 4.0 exists but was installed to a different location on your system, you need to modify the script and provide that path instead.

Compiling the C/C++ Code Samples

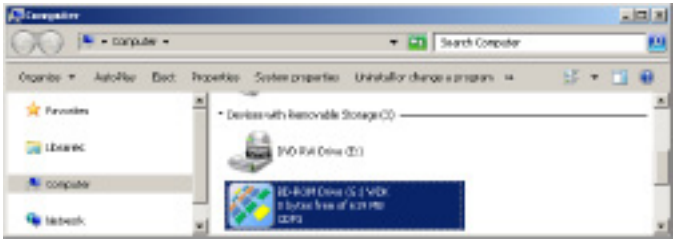
You can compile the C/C++ code samples from the companion source code using the Windows 7 Driver Development Kit (DDK) build tools. The following steps detail this process. I strongly recommend that you complete these steps before you start reading, because you'll need the code samples to follow the experiments shown later in this book.

1. Download the Windows 7 DDK ISO image from the Microsoft Download Center at *http://www.microsoft.com/download/en/details.aspx?id=11800*, and save it to your local hard drive. Set plenty of time aside for this download if you have a slow Internet connection; the DDK ISO file is over 600 MB in size.

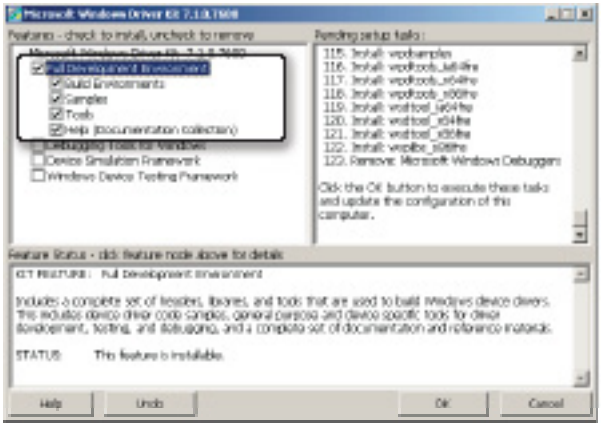
2. After the download is complete, mount the saved ISO file into a drive letter. There are several free tools for mounting ISO images on Windows. Virtual Clone Drive, which you can find on the Internet, is good freeware that works well both on Windows Vista and Windows 7. With that freeware installed, you should be able to right-click the ISO file and mount it, as shown in the following screen shot:



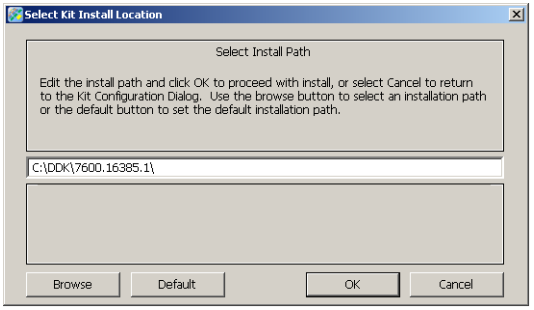
3. Double-click the newly mounted drive to kick off the DDK setup program, as shown here:



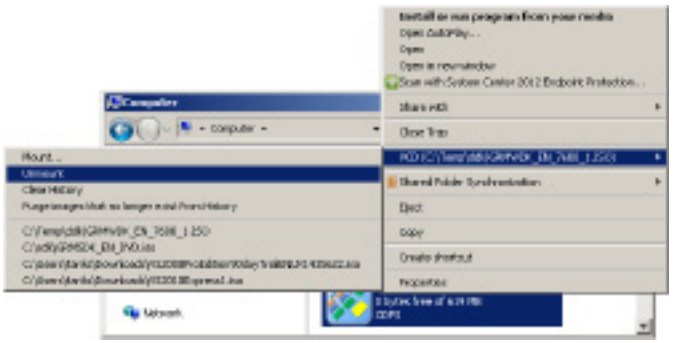
4. Select the Full Development Environment components from the DDK.



5. Then Install the components to the C:\DDK\7600.16835.1 directory, as shown in the following screen shot. This step will take several minutes to complete.



6. You can now unmount the DDK drive by right-clicking the mounted drive letter and selecting the Unmount menu action. This concludes this one-time installation of the Windows DDK build tools.



7. To build *x86* binaries, first start a command prompt window and type the following command:

```
C:\DDK\7600.16385.1>bin\setenv.bat c:\DDK\7600.16385.1
```

8. You can then build all the native code samples at once. Simply navigate to the root directory that you extracted the companion source code to and issue the following command. It should take only a minute or so to build all the C/C++ code from the companion source code:

```
C:\book\code>bcz
```

Acknowledgments

I am indebted to Vance Morrison for reviewing my rough drafts. Vance has been a role model for me since I joined Microsoft, and his critical insight was tremendously helpful in improving the quality of my writing. I only hope that I came close to his high standards.

I'd also like to thank Silviu Calinoiu and Shawn Farkas for their detailed review of my draft chapters. Their attention to detail and precise feedback was tremendously helpful.

Kalin Toshev provided valuable technical feedback during the formative stages of this book, and his unwavering dedication to test-driven software development really inspired me to write this book. This book owes a lot to him.

Ajay Bhave, Cristian Levcovici, and Rico Mariani provided several ideas for improving the material covered in this book. This book certainly wouldn't be the same without their help and guidance.

Special thanks to Michael Fortin for writing the foreword, and to all my colleagues in the Windows fundamentals team for their support during the writing of this book.

Finally, I have to acknowledge my family, who play an important part in my life. I would like to dedicate this book to my parents for providing me with the opportunity to reach for my dreams and always being there when I needed them. Your love and support mean the world to me.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/9780735662780

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

PART 1

A Bit of Background

One of my esteemed mentors at Microsoft once told me the following story. He came back home one night to be greeted by a dismayed look on his wife's face. Her cherished wedding ring had just vanished down the bathroom sink drain and she was completely stumped. For all she knew, the wedding ring might very well have been in the ocean by the time they were having that conversation. To her, anything beyond the drain plug was a black box. My mentor, using the slight advantage of plumbing knowledge he had over his wife, knew about those nifty structures called "sink traps," which are usually J-shaped pipes located just beneath the sink. These are intended to "trap" a little bit of water and prevent sewer gas from flowing out of the drain pipes into the living space. But in addition, the traps also conveniently capture objects and keep them from going down the drain immediately. As it turned out, the ring was indeed in the sink trap, and he was able to retrieve it relatively easily.

The reason my mentor told me this story was to illustrate a close parallel in the world of software engineering: If you treat the APIs and frameworks you use in your code as pure black boxes, you might be able to get by for a little while, but you are certainly bound to experience some pretty anxious moments when you need to investigate failures that fall just outside of your own code, even if the solution—just like in the lost ring analogy—is right under your nose.

The first part of this book explores how your programs interact with the Microsoft Windows operating system (in a loose sense) and demonstrates why it's useful to have at least a cursory understanding of both those interactions and of the role of each subsystem. It also examines some important development frameworks shipped by Microsoft, and analyzes their positions relative to each other and to the operating system (OS) developer interfaces. It then concludes by introducing the Windows Software Development Kit (SDK) tools and, more specifically, the Swiss Army knives of expert Windows developers—namely, the Windows debugger (WinDbg) and the Windows Performance Toolkit (Xperf). This knowledge will serve as a perfect segue to the rest of the book, where you'll use debugging and tracing to write better software for the Windows operating system.

CHAPTER 1

Software Development in Windows

In this chapter

Windows Evolution

Though this book focuses primarily on the post-Vista era of Windows, it's useful to look back at the history of Windows releases, because the roots of several building blocks of the underlying architecture can be traced all the way back to the Windows NT (an abbreviation for "New Technology") operating system. Windows NT was first designed and developed by Microsoft in the late '80s, and continued to evolve until its kernel finally became the core of all client and server versions of the Windows operating system.

Windows Release History

Windows XP marked a major milestone in the history of Windows releases by providing a unified code base for both the business (server) and consumer (client) releases of Windows. Though Windows XP was a client release (its server variant was Windows Server 2003), it technically succeeded both Windows 95/98/ME (a lineage of consumer operating systems that find their roots in the MS-DOS and Windows 3.1 operating systems) and Windows NT 4/Windows 2000, combining for the first time the power of the Windows NT operating system kernel and its robust architecture with many of the features that had made Windows 95 and Windows 98 instant hits with consumers and developers alike (friendly user design, aesthetic graphical interface, plug and play model, rich Win32 and DirectX API sets, and so on).

Though both the server and client releases of Windows now share the same kernel, they still differ in many of their features and components. (For example, only the server releases of Windows support multiple concurrent remote desktop user sessions.) Since the release of Windows XP in 2001, Windows Server has followed a release cycle that can be loosely mapped to corresponding Windows client releases. Windows Server 2003, for instance, shares many of the new kernel and API features that were added in Windows XP. Similarly, Windows Server 2008 R2 represents the server variant of Windows 7, which was released in late 2009. (Don't confuse this with Windows Server 2008, which is the server variant of Windows Vista.)

Figure 1-1 illustrates the evolution of the Windows family of operating systems, with their approximate release dates relative to each other.

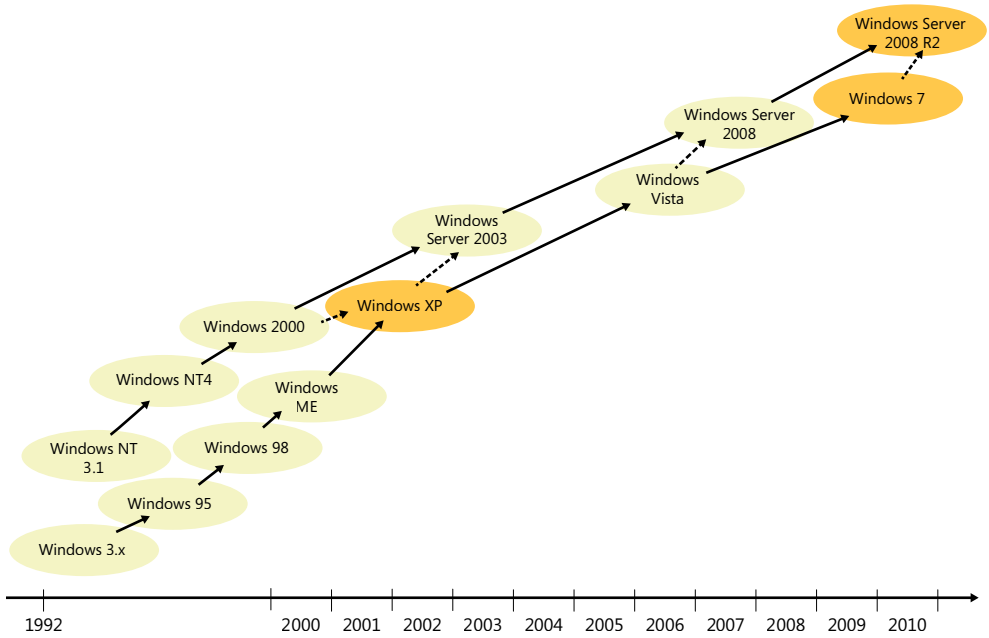


FIGURE 1-1 Timeline of major client and server releases of the Windows operating system since the early 90s.

Supported CPU Architectures

Windows was ported to many CPU architectures in the past. For example, Windows NT supported Alpha and MIPS processors until Windows NT 4. Windows NT 3.51 also had support for Power PC (another RISC family of processors that is used in many embedded devices, including, for example, Microsoft Xbox 360). However, Windows later narrowed its support to three CPU architectures: *x86* (a 32-bit family of processors, whose instruction set was designed by Intel), *x64* (also known as AMD-64, in reference to the fact this architecture was first introduced by AMD, though Intel also now releases processors implementing this instruction set), and *ia64* (another 64-bit instruction set designed by Intel in collaboration with Hewlett-Packard).

Microsoft shipped the first *ia64* version of Windows XP in late 2001 and followed it with an *x64* version in 2005. Microsoft later dropped support for *ia64* on client editions, including Windows XP. The *x86*, *x64*, and *ia64* architectures supported in Windows Server 2003 and Windows XP are exactly those that were also supported when Windows Server 2008 R2 and Windows 7 shipped at the end of 2009, though *x86* and *x64* are clearly the more widely used Windows architectures nowadays. Note, however, that Windows Server no longer supports *x86*; it now supports only 64-bit architectures. Also, Microsoft announced early in 2011 that its upcoming release of the Windows operating system will be capable of running on *ARM* (in addition to the *x86* and *x64* platforms), a RISC instruction set that's widely used in embedded utilities, smartphones, and tablet (slate) devices thanks in large part to its efficient use of battery power.

Understanding the underlying CPU architecture of the Windows installation you are working on is very important during debugging and tracing because you often need to use native tools that correspond to your CPU architecture. In addition, sometimes you will also need to understand the disassembly of the code you are analyzing in the debugger, which is different for each CPU. This is one reason many debugger listings in this book also show the underlying CPU architecture that they were captured on so that you can easily conduct any further disassembly inspection you decide to do on the right target platform. In the following listing, for example, the *vertarget* command shows a Windows 7 AMD64 (*x64*) operating system. You'll see more about this command and others in the next chapter, so don't worry about how to issue it for now.

```
lkd> $ Multi-processor (2 processors or cores) Windows 7 x64 system
lkd> vertarget
Windows 7 Kernel Version 7600 MP (2 procs) Free x64
Built by: 7600.16385.amd64fre.win7_rtm.090713-1255
```

Given the widespread use of *x86* and *x64*, and because you can also execute *x86* programs on *x64* machines, the majority of experiments in this book are conducted using the *x86* architecture so that you can follow them the way they're described in this book regardless of your target architecture. Though *x86* has been the constant platform of choice for Windows since its early days in the '80s, 64-bit processors continue to gain in popularity even among home computers and laptops, which now often carry *x64* versions of Windows 7.

Windows Build Flavors

The *vertarget* debugger command output shown in the previous section referred to the Windows version on the target machine as a "free" (also known as *retail*) build. This flavor is the only one ever shipped to end users by Microsoft for any of the supported processor architectures. There is, however, another flavor called a "checked" (also known as *debug*) build, which MSDN subscribers can obtain from Microsoft if they want to test the software they build with this flavor of the Windows operating system. It's important to realize that checked flavors are mostly meant to help driver developers; they don't derive their name at all from being "tested"—or otherwise "checked"—more thoroughly than the free flavors.

If you recall, the Introduction of this book recommended using the Driver Development Kit (DDK) build environment if you wanted to recompile the companion C++ sample code. As was explained then, you can also specify the build flavor you want to target (the default being *x86* free in the Windows 7 DDK) when starting a DDK build environment. This is, in fact, also how the checked flavor of Windows is built internally at Microsoft because the same build environment made available in the DDK is also used by Windows developers to compile the Windows source code. For example, the following command starts a DDK build environment where your source code is compiled into *x64* binaries using the checked (*chk*) build flavor. This essentially turns off a few compiler optimizations and defines debug build macros (such as the DBG preprocessor variable) that turn on "debug" sections of the code, including assertions (such as the NT_ASSERT macro).

```
C:\DDK\7600.16385.1>bin\setenv.bat c:\DDK\7600.16385.1 chk x64
```

Naturally, you don't really need a checked build of Windows to run your checked binaries, and the main difference between your "free" and "checked" binaries is that the assertions you put in your code will occur only in the "checked" flavor. The benefit of the checked flavor of Windows itself is that it also contains many additional assertions in the system code that can point out implementation problems in your code, which is usually useful if you are developing a driver. The drawback to that Windows flavor, of course, is that it runs much slower than the free flavor and also that you must run it with a kernel debugger attached at all times so that you can ignore assertions when they're hit; otherwise, those assertions might go unhandled and cause the machine to crash and reboot if they are raised in code that runs in kernel mode.

Windows Servicing Terminology

Each major Windows release is usually preceded by a few public milestones that provide customers with a preview of the features included in that release. Those prerelease milestones are usually called *Alpha*, *Beta1*, *Beta2,* and *RC* or release candidate, in this chronological order, though several Windows releases have either skipped some of these milestones or named them differently. These prerelease milestones also present an opportunity for Microsoft to engage with customers and collect their feedback before Windows is officially "released to manufacturing," a milestone referred to as *RTM*.

You will again recognize the major version of Windows in the build information displayed by the *vertarget* command that accompanies many of the debugger listings presented in this book. For example, the following listing shows that the target machine is running Windows 7 RTM and that July 13, 2009 (identified by the "090713" substring in the following output) is the date this particular Windows build was produced at Microsoft.

```
lkd> vertarget
Windows 7 Kernel Version 7600 MP (2 procs) Free x64
Built by: 7600.16385.amd64fre.win7_rtm.090713-1255
```

In addition to the major client and server releases for each Windows operating system, Microsoft also ships several servicing updates in between those releases that get automatically delivered via the Windows Update pipeline and usually come in one of the following forms:

- **Service packs** These releases usually occur a few years apart after RTM and compile the smaller updates made in between (and, on occasion, new features requested by customers) into a single package that can be applied at once by both consumers and businesses. They are often referred to using the "SP" abbreviation, followed by the service pack number. For example, SP1 is the first service pack after RTM, SP2 is the second, and so on.

  Service packs are considered major releases and are subjected to the same rigorous release process that accompanies RTM releases. In fact, many Windows service packs also have one or more release candidate (RC) milestones before they're officially released to the public. The number of service packs for a major Windows release is often determined by customer demand and also the amount of changes accumulated since the last service pack. Windows NT4, for example, had six service packs, while Windows Vista had two.

- **GDR updates** GDR (General Distribution Release) updates are issued to address bugs with broad impact or security implications. The frequency varies by need, but these updates are usually released every few weeks. These fixes are also rolled up into the following service pack release.

  For example, the following output indicates that the target debugging machine is running a version of Windows 7 SP1. Notice also that the version of *kernel32.dll* that's installed on this machine comes from a GDR update subsequent to the initial Windows 7 SP1 release.

```
0:000> vertarget
Windows 7 Version 7601 (Service Pack 1) MP (2 procs) Free x86 compatible
kernel32.dll version: 6.1.7601.17651 (win7sp1_gdr.110715-1504)
```

Windows Architecture

The fundamental design of the Windows operating system, with an executive that runs in kernel mode and a complementary set of user-mode system support processes (*smss.exe*, *csrss.exe*, *winlogon.exe*, and so on) to help manage additional system facilities, has for the most part remained unchanged since the inception of the Windows NT operating system back in the late '80s. Each new version of Windows naturally brings about a number of new components and APIs, but understanding how they fit in the architectural stack often starts with knowing how they interact with these core components of the operating system.

Kernel Mode vs. User Mode

Kernel mode is an execution mode in the processor that grants access to all system memory (including user-mode memory) and unrestricted use of all CPU instructions. This CPU mode is what enables the Windows operating system to prevent *user-mode* applications from causing system instability by accessing protected memory or I/O ports.

Application software usually runs in user mode and is allowed to execute code in kernel mode only via a controlled mechanism called a *system call*. When the application wants to call a system service exposed by code in the OS that runs in kernel mode, it issues a special CPU instruction to switch the calling thread to kernel mode. When the service call completes its execution in kernel mode, the operating system switches the thread context back to user mode, and the calling application is able to continue its execution in user mode.

Third-party vendors can get their code to run directly in kernel mode by implementing and installing signed *drivers*. Note that Windows is a monolithic system in the sense that the OS kernel and drivers share the same address space, so any code executing in kernel mode gets the same unrestricted access to memory and hardware that the core of the Windows operating system would have. In fact, several parts of the operating system (the NT file system, the TCP/IP networking stack, and so on) are also implemented as drivers rather than being provided by the kernel binary itself.

The Windows operating system uses the following layering structure for its kernel-mode operations:

- **Kernel** Implements core low-level OS services such as thread scheduling, multiprocessor synchronization, and interrupt/exception dispatching. The kernel also contains a set of routines that are used by the executive to expose higher-level semantics to user-mode applications.
- **Executive** Also hosted by the same "kernel" module in Windows (NTOSKRNL), and performs base services such as process/thread management and I/O dispatching. The executive exposes documented functions that can be called from kernel-mode components (such as drivers). It also exposes functions that are callable from user mode, known as *system services*. The typical entry point to these executive system services in user mode is the *ntdll.dll* module. (This is the module that has the system call CPU instruction!) During these system service calls, the executive allows user-mode processes to reference the objects (process, thread, event, and so on) it implements via indirect abstractions called object *handles*, which the executive keeps track of using a per-process handle table.
- **Hardware Abstraction Layer** The HAL (*hal.dll*) is a loadable kernel-mode module that isolates the kernel, executive, and drivers from hardware-specific differences. This layer sits at the very bottom of kernel layers and handles key hardware differences so that higher-level components (such as third-party device drivers) can be written in a platform-agnostic way.
- **Windows and Graphics Subsystem** The Win32 UI and graphics services are implemented by an extension to the kernel (*win32k.sys* module) and expose system services for UI applications. The typical entry point to these services in user mode is the *user32.dll* module.

Figure 1-2 illustrates this high-level architecture.

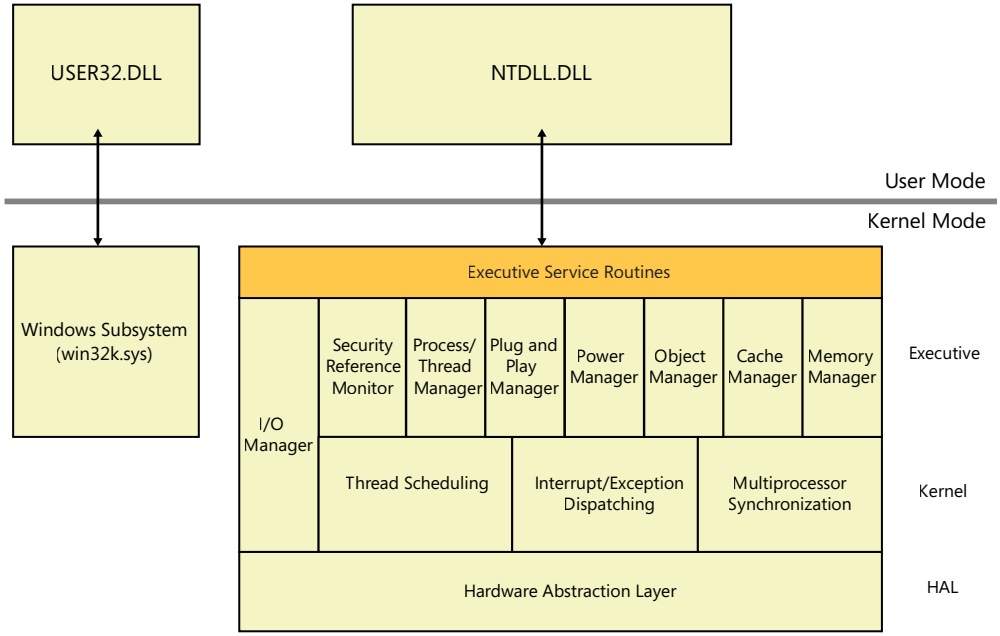


FIGURE 1-2 Kernel-mode layers and services in the Windows operating system.

User-Mode System Processes

Several core facilities (logon, logoff, user authentication, and so on) of the Windows operating system are primarily implemented in user mode rather than in kernel mode. A fixed set of user-mode system processes exists to complement the OS functionality exposed from kernel mode. Here are a few important processes that fall in this category:

- ***Smss.exe*** User sessions in Windows represent resource and security boundaries and offer a virtualized view of the keyboard, mouse, and physical display to support concurrent user logons on the same OS. The state that backs these sessions is tracked in a kernel-mode virtual memory space usually referred to as the *session space*. In user mode, the session manager subsystem process (*smss.exe*) is used to start and manage these user sessions.

  A "leader" *smss.exe* instance that's not associated with any sessions gets created as part of the Windows boot process. This leader *smss.exe* creates a transient copy of itself for each new session, which then starts the *winlogon.exe* and *csrss.exe* instances corresponding to that user session. Although having the leader session manager use copies of itself to initialize new sessions doesn't provide any practical advantages on client systems, having multiple *smss.exe* copies running concurrently can provide faster logon of multiple users on Windows Server systems acting as Terminal Servers.

- ***Winlogon.exe*** The Windows logon process is responsible for managing user logon and logoff. In particular, this process starts the logon UI process that displays the logon screen when the user presses the Ctrl+Alt+Del keyboard combination and also creates the processes

responsible for displaying the familiar Windows desktop after the user is authenticated. Each session has its own instance of the *winlogon.exe* process.

- ***Csrss.exe*** The client/server runtime subsystem process is responsible for the user-mode portion of the Win32 subsystem (*win32k.sys* being the kernel-mode portion) and also was used to host the UI message loop of console applications prior to Windows 7. Each user session has its own instance of this process.

- ***Lsass.exe*** The local security authority subsystem process is used by *winlogon.exe* to authenticate user accounts during the logon sequence. After successful authentication, LSASS generates a security access token object representing the user's security rights, which are then used to create the new explorer process for the user session. New child processes created from that shell then inherit their access tokens from the initial explorer process security token. There is only one single instance of this process, which runs in the noninteractive session (known as session 0).

- ***Services.exe*** This system process is called the *NT service control manager* (SCM for short) and runs in session 0 (noninteractive session). It's responsible for starting a special category of user-mode processes called *Windows services*. These processes are generally used by the OS or third-party applications to carry out background tasks that do not require user interaction. Examples of Windows services include the spooler print service (*spooler*); the task scheduler service (*schedule*); the COM activation services, also known as the COM SCM (*RpcSs* and *DComLaunch*); and the Windows time service (*w32time*).

  These processes can choose to run with the highest level of user-mode privileges in Windows (*LocalSystem* account), so they are often used to perform privileged tasks on behalf of user-mode applications. Also, because these special processes are always started and stopped by the SCM process, they can be started on demand and are guaranteed to have at most one active instance running at any time.

All of the aforementioned system-support processes run under the *LocalSystem* account, which is the highest privileged account in Windows. Processes that run with this special account identity are said to be a part of the trusted computing base (TCB) because once user code is able to run with that level of privilege, it is also able to bypass any checks by the security subsystem in the OS.

User-Mode Application Processes

Every user-mode process (except for the leader *smss.exe* process mentioned earlier) is associated with a user session. These user-mode processes are boundaries for a memory address space. As far as scheduling in Windows is concerned, however, the most fundamental scheduling units remain the threads of execution and processes are merely containers for those threads. It's also important to realize that user-mode processes (more specifically, the threads they host) also often run plenty of code in kernel mode. Although your application code might indeed run in user mode, it's often the case that it also calls into system services (through API layers that call down to NTDLL or USER32 for the system call transitions) that end up transitioning to kernel mode on your behalf. This is why it makes sense to always think of your software (whether it's user-mode software or kernel drivers) as

an extension of the Windows operating system and also that you understand how it interacts with the "services" provided by the OS.

Processes, in turn, can be placed in containers called *job objects*. These executive objects can be very useful to manage a group of processes as a single unit. Unlike threads and processes, job objects are often overlooked when studying the Windows architecture despite their unique advantages and the useful semantics they provide. Figure 1-3 illustrates the relationship between these fundamental objects.

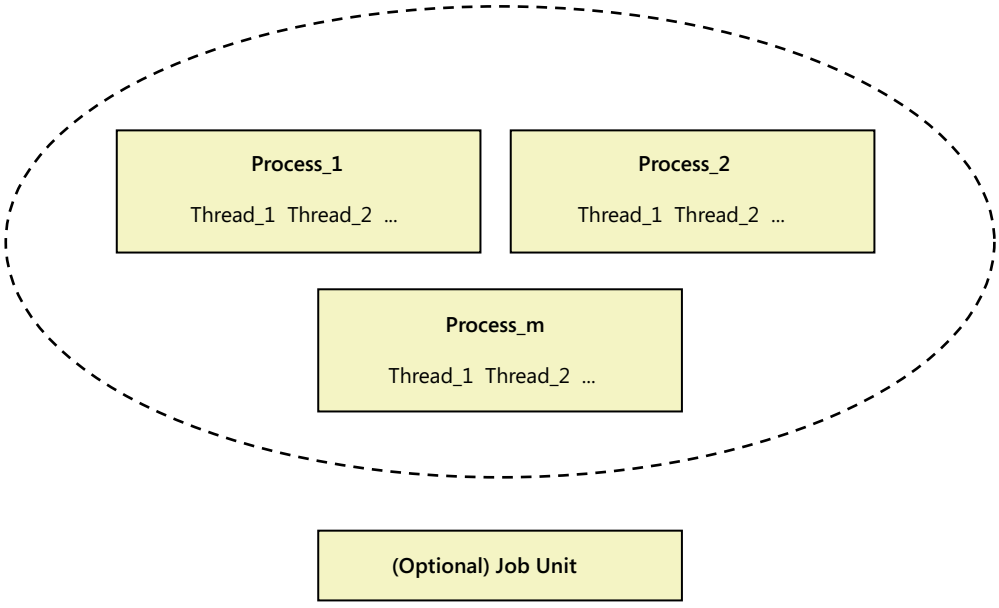


FIGURE 1-3 Threads, processes, and jobs in Windows.

Job objects can be used to provide common execution settings for a set of processes and, among other things, to control the resources used by member processes (such as the amount of memory consumed by the job and the processors used for its execution) or their UI capabilities.

One particularly useful feature of job objects is that they can be configured to terminate their processes when their user-mode job handle is closed (either using an explicit *kernel32!CloseHandle* API call, or implicitly when the kernel runs down the handles in the process handle table when the process kernel object is destroyed). To provide a practical illustration, the following C++ program shows how to take advantage of the job-object construct exposed by the Windows executive in a C++ user-mode application to start a child ("worker") process and synchronize its lifetime with that of its parent process. This is often useful in the case of worker processes whose sole purpose is to serve requests in the context of their parent process, in which case it becomes critical not to "leak" those worker instances should the parent process die unexpectedly. (The reverse is more straightforward because the parent process can easily monitor when the child dies by simply waiting on the worker process handle to become signaled using the *kernel32!WaitForSingleObject* Win32 API.)

To follow this experiment, remember to refer back to the Introduction of this book, which contains step-by-step instructions for how to build the companion source code.

```
//
// C:\book\code\chapter_01\WorkerProcess>main.cpp
//
class CMainApp
{
public:
    static
    HRESULT
    MainHR()
    {
        HANDLE hProcess, hPrimaryThread;
        CHandle shProcess, shPrimaryThread;
        CHandle shWorkerJob;
        DWORD dwExitCode;
        JOBOBJECT_EXTENDED_LIMIT_INFORMATION exLimitInfo = {0};
        CStringW shCommandLine = L"notepad.exe";

        ChkProlog();

        //
        // Create the job object, set its processes to terminate on
        // handle close (similar to an explicit call to TerminateJobObject),
        // and then add the current process to the job.
        //
        shWorkerJob.Attach(CreateJobObject(NULL, NULL));
        ChkWin32(shWorkerJob);

        exLimitInfo.BasicLimitInformation.LimitFlags =
            JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE;
        ChkWin32(SetInformationJobObject(
            shWorkerJob,
            JobObjectExtendedLimitInformation,
            &exLimitInfo,
            sizeof(exLimitInfo)));

        ChkWin32(AssignProcessToJobObject(
            shWorkerJob,
            ::GetCurrentProcess()));

        //
        // Now launch the new child process (job membership is inherited by default)
        //
        wprintf(L"Launching child process (notepad.exe) ...\n");
        ChkHr(LaunchProcess(
            shCommandLine.GetBuffer(),
            0,
            &hProcess,
            &hPrimaryThread));
```

```
            shProcess.Attach(hProcess);
            shPrimaryThread.Attach(hPrimaryThread);

            //
            // Wait for the worker process to exit
            //
            switch (WaitForSingleObject(shProcess, INFINITE))
            {
                case WAIT_OBJECT_0:
                    ChkWin32(::GetExitCodeProcess(shProcess, &dwExitCode));
                    wprintf(L"Child process exited with exit code %d.\n", dwExitCode);
                    break;
                default:
                    ChkReturn(E_FAIL);
            }

            ChkNoCleanup();
        }
    };
```

One key observation here is that the parent process is assigned to the new job object before the new child process is created, which allows the worker process to automatically inherit this job membership. This means in particular that there is no time window in which the new process would exist without being a part of the job object. If you kill the parent process (using the Ctrl+C signal, for example), you will notice that the worker process (*notepad.exe* in this case) is also terminated at the same time, which was precisely the desired behavior.

```
C:\book\code\chapter_01\WorkerProcess>objfre_win7_x86\i386\workerprocess.exe
Launching child process (notepad.exe) ...
^C
```

Low-Level Windows Communication Mechanisms

With code executing in kernel and user modes, and also inside the boundaries of per-process address spaces in user mode, the Windows operating system supports several mechanisms for allowing components to communicate with each other.

Calling Kernel-Mode Code from User Mode

The most basic way to call kernel-mode code from user-mode components is the *system call* mechanism mentioned earlier in this chapter. This mechanism relies on native support in the CPU to implement the transition in a controlled and secure manner.

One inherent drawback to the system call mechanism is that it relies on a hard-coded table of well-known executive service routines to dispatch the request from the client code in user mode to its intended target service routine in kernel mode. This doesn't extend well to kernel extensions implemented in the form of drivers, however. For those cases, another mechanism—called *I/O control*

commands (*IOCTL*)—is supported by Windows to enable user-mode code to communicate with kernel-mode drivers. This is done through the generic *kernel32!DeviceIoControl* API, which takes the user-defined IOCTL identifier as one of its parameters and also a handle to the device object to which to dispatch the request. The transition to kernel mode is still performed in the NTDLL layer (*ntdll!NtDeviceIoControlFile*) and internally also uses the system call mechanism. So, you can think of the IOCTL method as a higher-level user/kernel communication protocol built on top of the raw system call services provided by the OS and CPU.

Internally, I/O control commands are processed by the I/O manager component of the Windows executive, which builds what is called an *I/O request packet* (*IRP* for short) that it then routes to the device object requested by the caller from user mode. IRP processing in the Windows executive uses a layered model where devices have an associated driver stack that handles their requests. When an IRP is sent to a top-level device object, it travels through its device stack starting at the top, passing through each driver in the corresponding device stack and giving it a chance to either process or ignore the command. In fact, IRPs are also used in kernel mode to send commands to other drivers so that the same IRP model is used for interdriver communication in the kernel. Figure 1-4 depicts this architecture.

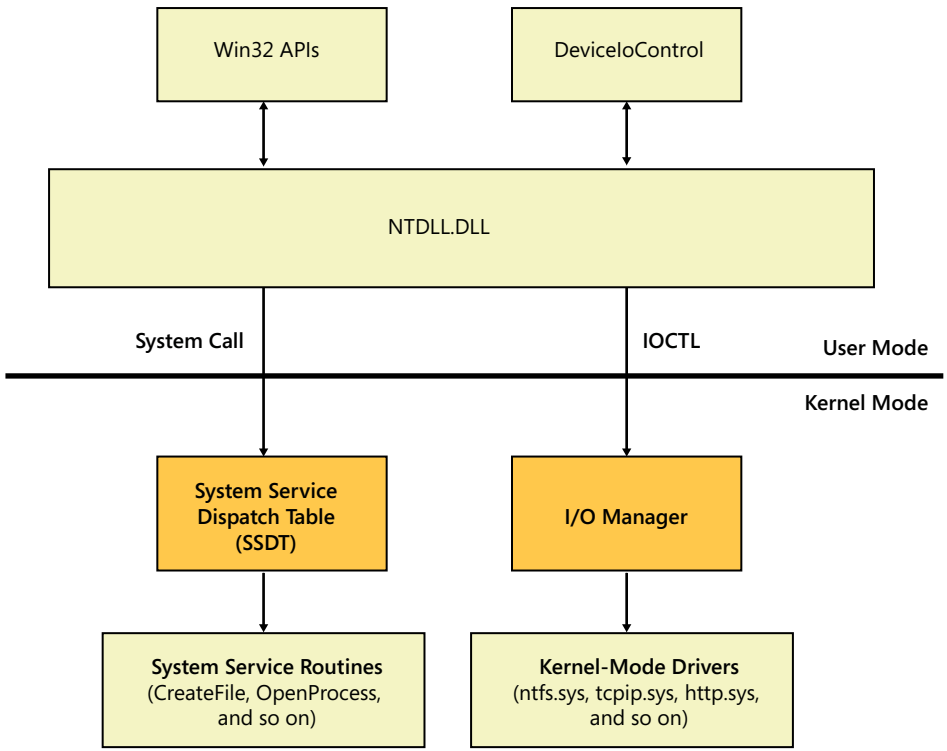


FIGURE 1-4 User-mode to kernel-mode communication mechanisms.

Calling User-Mode Code from Kernel Mode

Code that runs in kernel mode has unrestricted access to the entire virtual address space (both the user and kernel portions), so kernel mode in theory could invoke any code running in user mode. However, doing so requires first picking a thread to run the code in, transitioning the CPU mode back to user mode, and setting up the user-mode context of the thread to reflect the call parameters. Fortunately, however, only the system code written by Microsoft really needs to communicate with random threads in user mode. The drivers you write, on the other hand, need to call back to user mode only in the context of a device IOCTL initiated by a user-mode thread, so they do not need a more generic kernel-mode to user-mode communication mechanism.

A standard way for the system to execute code in the context of a given user-mode thread is to send an asynchronous procedure call (APC) to that thread. For example, this is exactly how thread suspension works in Windows: the kernel simply sends an APC to the target thread and asks it to execute a function to wait on its internal thread semaphore object, causing it to become suspended. APCs are also used by the system in many other scenarios, such as in I/O completion and thread pool callback routines, just to cite a couple.

Interprocess Communication

Another way for communicating between user-mode processes and code in kernel mode, as well as between user-mode processes themselves, is to use the advanced local procedure call (ALPC) mechanism. ALPC was introduced in the Windows Vista timeframe and is a big revision of the LPC mechanism, a feature that provided in many ways the bloodline of low-level intercomponent communication in Windows since its early releases.

ALPC is based on a simple idea: a server process first opens a kernel port object to receive messages. Clients can then connect to the port if allowed by the server owning the port and start sending messages to the server. They are also able to wait until the server has fetched and processed the message from the internal queue that's associated with the ALPC port object.

In the case of user/user ALPC, this provides a basic low-level interprocess communication channel. In the case of kernel/user ALPC channels, this essentially provides another (indirect) way for user-mode applications to call code in kernel mode (whether it's in a driver or in the kernel module itself) and vice versa. An example of this communication is the channel that's established between the *lsass.exe* user-mode system process and the security reference monitor (SRM) executive component in kernel mode, which is used, for example, to send audit messages from the executive to *lsass.exe*. Figure 1-5 illustrates this architecture.

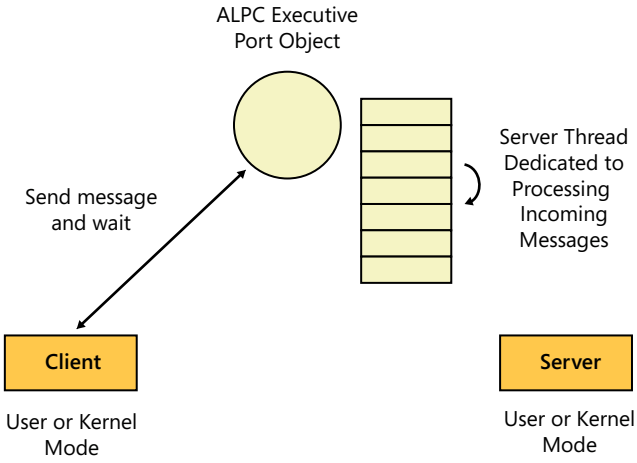


FIGURE 1-5 ALPC communication in Windows.

ALPC-style communication is used extensively in the operating system itself, most notably as it pertains to this book to implement the low-level communication protocol that native user-mode debuggers employ to receive various debug events from the process they debug. ALPC is also used as a building block in higher-level communication protocols such as local RPC, which in turn is used as the transport protocol in the COM model to implement interprocess method invocations with proper parameter marshaling.

Windows Developer Interface

Developers can extend the services that ship with the Windows operating system by building extensions in the way of kernel drivers or standalone user-mode Windows applications. This section examines some of the key layers and APIs that make building these extensions possible.

Developer Documentation Resources

Microsoft documents several APIs that developers can use when building their applications. The difference between these published interfaces and the internal (private) implementation details of the platform is that Microsoft has committed over the past two decades to building an ecosystem in which the public interfaces are carried forward with new releases of Windows, affording application developers the confidence that the applications they build today will continue to work on future OS versions. It's fair to say that this engineering discipline is one of the reasons Windows has been so successful with developers and end users alike.

Microsoft documents all of the interfaces and APIs it publishes on the Microsoft Developer Network (MSDN) website at *http://www.microsoft.com/msdn*. When writing your software, you should use only officially supported public APIs so that your software doesn't break when new versions of the operating system are released by Microsoft. Undocumented APIs can often disappear or get

renamed, even in service pack releases of the same OS version, so you should never target them in your software unless you are prepared to deal with the wrath of your customers when your software ominously stops working after a new Windows update is installed. That's not to say you shouldn't be interested in internal implementation details, of course. In fact, this book proves that knowing some of those details is often important when debugging your programs and can help you analyze their behaviors more proficiently, which in turns also helps you write better and more reliable software in the long run.

In addition to documenting the public APIs written by Microsoft for developers, the MSDN website also contains a wealth of articles describing features or areas at a higher level. In particular, it hosts a special category of articles, called *Knowledge Base articles* (KB articles for short) that are published by Microsoft's customer support team to document workarounds for known issues. Generally speaking, the MSDN website should be your first stop when looking up the documented behavior of the APIs you use in your code or when trying to learn how to use a new feature in the OS.

WDM, KMDF, and UMDF

Developers can run their code with kernel-mode privileges and extend the functionality of the OS by implementing a kernel-mode driver. Though the vast majority of developers will never need to write a kernel driver, it's still useful to understand the layered plug-in models used by Windows to support these kernel extensions because this knowledge sometimes can help you make sense of kernel call stacks during kernel-mode debugging investigations.

Driver extensions are often needed to handle communication with hardware devices that aren't already supported because—as mentioned earlier—user-mode code isn't allowed to access I/O ports directly. In addition, drivers are sometimes used to implement or extend system features. For example, many tools in the SysInternals suite—including the process monitor tool, which installs a filter driver to monitor access to system resources—internally install drivers when they're executed to implement their functionality.

There are many ways to write drivers, but the main model used to implement them in Windows is the Windows Driver Model (WDM). Because this model asks a lot from driver developers in terms of handling all the interactions with the I/O manager and the rest of the operating system and often results in a lot of duplicated boilerplate code that has to be implemented by all driver developers, the kernel-mode driver framework (KMDF) was introduced to simplify the task of writing kernel-mode drivers. Keep in mind, however, that KMDF doesn't replace WDM; rather, it's a framework that helps you more easily write drivers that comply with WDM's requirements. Generally speaking, you should write your drivers using KMDF unless you find a good reason not to do so, such as when you need to write non-WDM drivers. This is the case, for instance, for network, SCSI, or video drivers, which have their own world, so to speak, and require you to write what is called a "miniport" driver to plug into their respective port drivers.

A subset of hardware drivers also can be executed completely in user mode (though without direct access to kernel memory space or I/O ports). These drivers can be developed using another framework shipped by Microsoft called the user-mode driver framework, or UMDF. For more details on

the different driver models and their architectures, you can find a wealth of resources on the MSDN website at *http://msdn.microsoft.com*. The OSR website at *http://www.osronline.com* is also worth a visit if you ever need to write or debug drivers in Windows.

The NTDLL and USER32 Layers

As mentioned earlier in this chapter, the NTDLL and USER32 layers contain the entry points to the executive service routines and kernel-mode portion of the Win32 subsystem (*win32k.sys*), respectively.

There are hundreds of executive service stubs in the NTDLL module (*ntdll!NtSetEvent*, *ntdll!NtReadFile*, and many others). The majority of these service stubs are undocumented in the MSDN, but a few stub entry points were deemed generally useful to third-party system software and are documented. The NTDLL.DLL system module also hosts several low-level OS features, such as the module loader (*ntdll!Ldr** routines), the Win32 subsystem process communication functions (*ntdll!Csr** routines), and several run-time library functions (*ntdll!Rtl** routines) that expose features such as the Windows heap manager and Win32 critical section implementations.

The NTDLL module is used by many other DLLs in the Win32 API to transition into kernel mode and call executive service routines. Similarly, the USER32 DLL is also used by the Windows graphics architectural stack (DirectX, GDI32, and so on) as the gateway to transition into kernel mode so that it can communicate with the graphics processing unit (GPU) hardware.

The Win32 API Layer

The Win32 API layer is probably the most important layer to learn for developers who are new to Windows because it's the official public interface to the services exposed by the operating system. All of the Win32 API functions are documented in the MSDN, along with their expected parameters and potential return codes. Even if you are writing your software using a higher-level development framework or API set, as most of us do these days, being aware of the capabilities exposed at this layer will help you get a much better feel for a framework's advantages and limitations relative to other choices, as well as the raw capabilities exposed by the Win32 API and Windows executive.

The Win32 API layer covers a large set of functionality, going from basic services like creating threads/processes or drawing shapes on the screen to higher-level areas such as cryptography. The most basic services at the bottom of the Win32 API's architectural stack are exposed in the *kernel32.dll* module. Other widely used Win32 DLL modules are *advapi32.dll* (general utility functions), *user32.dll* (Windows and user object functions), and *gdi32.dll* (graphics functions). In Windows 7, the Win32 DLL modules are now layered so that lower-level base functions aren't allowed to call up to higher-level modules in the hierarchical stack. This layering engineering discipline helps prevent circular dependencies between modules and also minimizes the performance impact of bringing a new DLL dependency from the Win32 API set into your process address space. This is why you will see that many of the public APIs exported in the *kernel32.dll* module now simply forward their calls to the implementation defined in the lower-level *kernelbase.dll* DLL module, which is useful to know when trying to set system breakpoints in the debugger. This layered hierarchy is demonstrated in Figure 1-6.

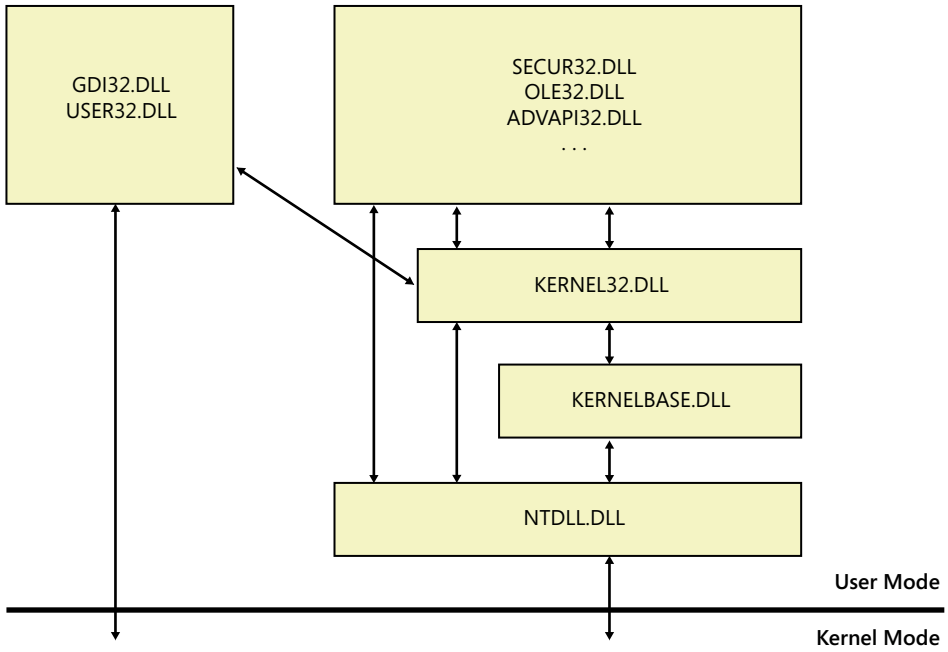


FIGURE 1-6 Low-level Win32 DLL modules.

The COM Layer

The Component Object Model (COM) was introduced by Microsoft in the mid-90s as a user-mode framework to enable developers to write reusable object-oriented components in different programming languages. If you are new to COM, the best resource to use to get started and gain an understanding of its model is the "Component Object Model Specification" document, which you can still find online. Though this is an old (circa 1995) and relatively long document, it provides a great overview of the COM binary specification, and much of what it describes still holds true to this day.

Over time, the use of the term "COM" grew to cover a number of different but related technologies, including the following:

- The object model itself, which the label "COM" was technically designed to describe at first. Key parts of this object model are the standard *IUnknown* and *IClassFactory* interfaces, the idea of separation of class interfaces (the public contract) from the internal COM class implementation, and the ability to query a server object for an implementation of the contracts that it supports (*IUnknown::QueryInterface*). This model, in its pure form, is one of the most elegant contributions by Microsoft to the developer ecosystem, and it has had a deep impact that still reverberates to this day in the form of various derivative technologies based on that model.
- The interprocess communication protocol and registration that allows components to communicate with each other without the client application having to know where the server component lives. This enables COM clients to transparently use servers implemented

locally in-process (DLLs) or out-of-process (EXEs), as well as servers that are hosted on a remote computer. The distinction between COM and Distributed COM (DCOM, or COM across machines) is often only theoretical, and most of the internal building blocks are shared between the two technologies.

- The protocol specifications built on top of the COM object model to enable hosts to communicate with objects written in multiple languages. This includes the OLE Automation and ActiveX technologies, in particular.

COM in the Windows Operating System

COM is omnipresent in the Windows operating system. Although the Microsoft .NET Framework has largely superseded COM as the technology of choice for Windows application-level development, COM has had a lasting impact on the Windows development landscape, and it's far from a dead technology as it continues to be the foundation for many components (for example, the Windows shell UI uses COM extensively) and even some of the newest technologies in the Windows development landscape. Even on a Windows installation with no other additional applications, you will still find thousands of COM class identifiers (CLSID for short) describing existing COM classes (servers) in the COM registry catalog, as shown in Figure 1-7.

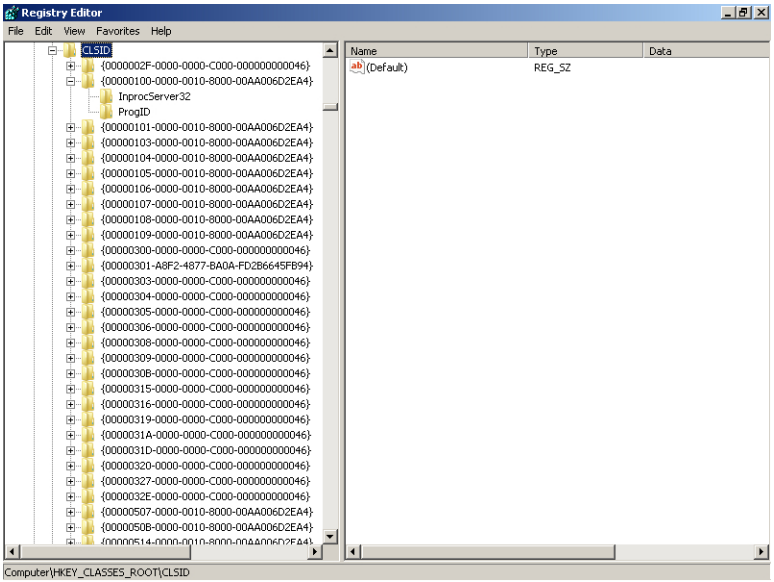


FIGURE 1-7 COM CLSID hive in the Windows registry.

More recently, the new model unveiled by Microsoft for developing touch-capable, Windows runtime (WinRT) applications in its upcoming version of Windows also uses COM for its core binary compatibility layer. In many ways, COM remains required knowledge if you really want to master the intricacies of Windows as a development platform. In addition, its design patterns and programming model have an educational value of their own. Even if you are never going to write a COM object (server), the programs and scripts you write often use existing COM objects, either explicitly or

indirectly via API calls. So, knowing how COM works is particularly useful in debugging situations that involve such calls.

COM developers usually interact primarily with language-level features and tools when writing or consuming COM servers. However, debugging COM failures also requires knowledge of how the system implements the COM specification. In that sense, there are a few different aspects to the COM landscape that can all come into play during COM debugging investigations:

- **The COM "Library"** This is essentially Microsoft's system implementation of the COM binary specification. The COM library consists primarily of the COM run-time code and Win32 APIs (*CoInitialize*, *CoCreateInstance*, and others) that ship as part of the *ole32.dll* system module. This run-time code also uses support provided by two Windows services collectively referred to as the *COM service control manager* (*COM SCM*). These services are the *RpcSs* service, which runs with *NetworkService* privileges, and the *DComLaunch* service, which runs with *LocalSystem* privileges.
- **COM language tools** These are the source-level compilers and tools to support COM's binary specification. This includes the interface definition language (IDL) compiler (MIDL), which allows COM classes and interfaces to be consumed inside C/C++ programs, and the binary type library importer and exporter tools, which enable cross-language interoperability in COM. Note that these tools do not ship with the OS, but come as part of developer tools such as the Windows SDK.
- **COM frameworks** These are frameworks that make it easier for developers to write COM components that conform to the COM binary specification. An example is the Microsoft C++ Active Template Library (ATL).

Writing COM Servers

The COM specification places several requirements on developers writing COM objects and their hosting modules. Writing a COM object in C++ entails, at the very least, declaring its published interfaces in an IDL file and implementing the standard *IUnknown* COM interface that allows reference counting of the object and enables clients to negotiate contracts with the server. A class factory object—a COM-style support object that implements the *IClassFactory* COM interface but doesn't need to be published to the registry—must also be written for each CLSID to create its object instances.

On top of all of this, the developer is also required to implement the hosting module (DLL or EXE) so that it also conforms to all the other requirements of the COM specification. For example, in the case of a DLL module, a C-style function (*DllGetClassObject*) that returns a pointer to the class factory of CLSIDs hosted by the module must be exported for use by the COM library. For an executable module, the COM library can't simply call an exported function, so *ole32!CoRegisterClassObjects* must be called by the server executable itself when it starts up in order to publish its hosted COM class factories and make COM aware of them. Yet another requirement for DLL COM modules is to implement reference counting of their active objects and export a C-style function (*DllCanUnloadNow*) so that the COM library knows when it's safe to unload the module in question.

Microsoft realized that this is a lot to ask of C++ COM developers and introduced the Active Template Library (ATL) to help simplify writing COM server modules and objects in the C++ language. Although the majority of C++ COM developers in Windows use ATL to implement their COM servers, keep in mind that you can also write COM objects and their hosting modules without it (if you feel inclined to do so). In fact, ATL ships with the source code of its template classes, so you can study those implementation headers and see how ATL implements its functionality on top of the COM services provided by the COM library in the operating system. As you'll see after looking at that source code, ATL takes care of much of the heavy lifting and boilerplate code for writing COM servers so that you don't have to. This allows you to focus on writing the code that implements your business logic without much of the burden imposed by the necessary COM model plumbing.

Managing Module Lifetimes in COM Servers

To give you a practical scenario so that you get a better feel for the type of features ATL provides, consider the problem of managing the lifetime of COM servers. As mentioned earlier, COM servers implement reference counting to determine when their hosting server module (DLL or EXE) can be unloaded safely. In the COM ATL library, for example, every time a new COM object is created within a COM module (*CAtlDllModule*), a global (per-module) integer variable counting the number of live objects implemented by that module is incremented by one. This special reference count is often referred to as the COM module's *lock count*. When the last reference to the object is released, the lock count is decremented by one. In addition, a client is also allowed to put a lock on the server (that is, increment its global lock count by one) by calling the *LockServer* method of the *IClassFactory* standard COM interface (implemented by the *CComClassFactory* class in ATL).

In the case of DLL COM servers, this essentially provides a simple implementation to ref-count the COM module so that when its *DllCanUnloadNow* function is called, the module is able to see the number of outstanding client references it has (the lock count) and, when that number drops to zero, report to the COM library that it's free to unload the DLL if it wants to. A host process is able to force this cleanup at any time by calling the *ole32!CoFreeUnusedLibraries* COM runtime function, which internally invokes the exported *DllCanUnloadNow* functions from all in-process DLL COM server modules loaded in the client process and unloads the modules that return TRUE.

The lock count is also used by COM EXE servers. In the case of ATL, for example, COM EXE modules (*CAtlExeModule*) use an extra thread that periodically checks their lock count and shuts the process down after a certain period of inactivity after its lock count drops to zero. You can see this logic in the ATL implementation headers, as in the *MonitorShutdown* callback shown in the following listing.

```
//
// c:\ddk\7600.16385.1\inc\atl71>atlbase.h
//
void MonitorShutdown() throw()
{
```

```
    while (1)
    {
        ::WaitForSingleObject(m_hEventShutdown, INFINITE);
        DWORD dwWait = 0;
        do        {
            m_bActivity = false;
            dwWait = ::WaitForSingleObject(m_hEventShutdown, m_dwTimeOut);
        } while (dwWait == WAIT_OBJECT_0);

        if (!m_bActivity && m_nLockCnt == 0)
        {
            ::CoSuspendClassObjects();
            if (m_nLockCnt == 0)
                break;
        }
    }
    ::CloseHandle(m_hEventShutdown);
    ::PostThreadMessage(m_dwMainThreadID, WM_QUIT, 0, 0);
}
```

Note that COM as an OS component and platform leaves implementing this module reference counting scheme up to the developer writing the COM server. Fortunately, however, the good folks at Microsoft who wrote the ATL framework took care of this on behalf of all the Windows developers writing COM servers using that framework.

Consuming COM Objects

Communication between COM clients and servers is a two-step process:

- **COM activation** This is the first step in the communication, where the COM library locates the class factory of the requested CLSID. This is done by first consulting the COM registry catalog to find the module that hosts the implementation of the COM server. The COM client code initiates this step with a call to either the *ole32!CoCreateInstance* or *ole32!CoGetClassObject* Win32 API call. Note that *ole32!CoCreateInstance* is simply a convenient wrapper that first calls *ole32!CoGetClassObject* to obtain the class factory, and then invokes the *IClassFactory::CreateInstance* method implemented by that class factory object to finally create a new instance of the target COM server.
- **Method invocations** After the COM activation step retrieves a proxy or direct pointer to the COM class object, the client can then query the interfaces exposed by the object and directly invoke the exposed methods.

A key point to understand when consuming COM objects in your code is the potential involvement of the COM SCM behind the scenes to instantiate the hosting module for the COM object and its corresponding class factory during the COM activation step. This is done because COM clients sometimes need to activate out-of-process servers in different contexts, such as when the COM server object needs to run in processes with higher privileges or in different user sessions, which requires the participation of a broker process that runs with higher privileges (the *DComLaunch* service). Another

reason is that the *RpcSs* Windows service also handles cross-machine COM activation requests (the DCOM case) and implements the communication channel in a way that's completely transparent to both COM clients and servers. It's especially important to understand this involvement during debugging investigations of COM activation failures. Once the COM activation sequence retrieves the requested class factory, however, the COM client is then able to directly invoke COM methods published by the server class without any involvement on the part of the COM SCM. Figure 1-8 summarizes these steps and the key components involved during the COM activation sequence.

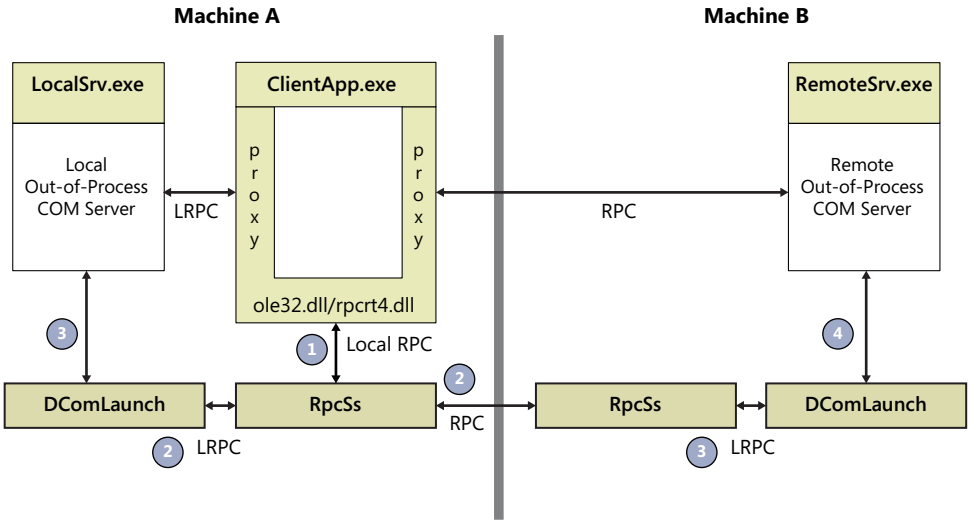


FIGURE 1-8 COM activation components.

The nice thing about the COM model, as mentioned earlier in this section, is that the client doesn't need to know where the COM server implementation lives, or even the language (C++, Microsoft Visual Basic, Delphi, or other) in which it was written (provided it has access to a type library or a direct virtual table layout that describes the COM types it wants to consume). The only thing that the client needs to know is the CLSID of the COM object (a GUID), after which it can query for the supported interfaces and invoke the desired methods. COM in the OS provides all the necessary "glue" for the client/server communication, provided the COM server was written to conform to the COM model. In particular, COM supports accessing the following COM server types using the same consistent programmatic model:

- **In-process COM servers** The hosting DLL module is loaded into the client process address space, and the object is invoked through a pointer returned by the COM activation sequence. This pointer can be either a direct virtual pointer or sometimes a proxy, depending on whether the COM runtime needs to be invoked to provide additional safeguards (such as thread safety) before invoking the methods of the COM server.
- **Local/remote out-of-process COM servers** For local out-of-process COM servers, local RPC is used as the underlying interprocess communication protocol, with ALPC as the actual low-level foundation. For remote COM servers (DCOM), the RPC communication protocol is used

for the intermachine communication. In both cases, the proxy memory pointer that is returned to the client application from the COM activation sequence takes care of everything that's required to accomplish COM's promise of transparent remoting. Figure 1-9 illustrates this aspect.

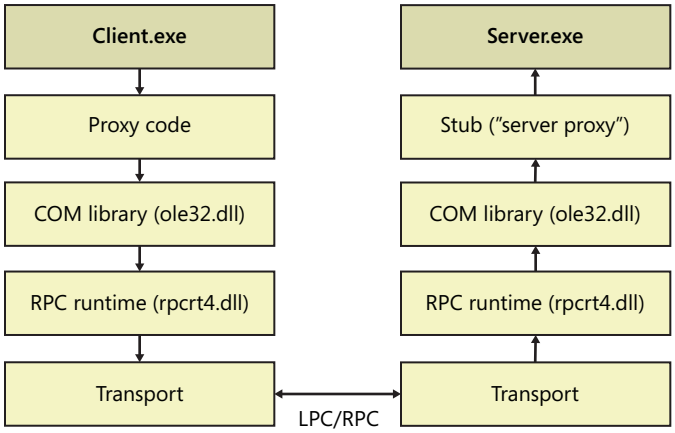


FIGURE 1-9 Out-of-process COM method invocations.

The CLR (.NET) Layer

Like COM, the .NET Framework is also a user-mode, object-oriented platform that enables developers to write their programs in their language of choice (C#, Microsoft Visual Basic .NET, C++/CLI, or other). However, .NET takes another leap and has those programs run under the control of an execution engine environment that provides additional useful features, such as type safety and automatic memory management using garbage collection. This execution engine environment is called the *Common Language Runtime* (*CLR*) and is in many ways the core of the .NET platform. Understanding this layer is often helpful when debugging applications built on top of the various .NET class libraries and technologies (ASP.NET, WCF, WinForms, WPF, Silverlight, and so on).

The CLR runtime is implemented as a set of native DLLs that get loaded into the address space of every .NET executable, though the core execution engine DLL decides when to load the other DLL dependencies. Because of their reliance on this execution environment, .NET modules (also called *assemblies*) are said to be *managed*, as opposed to the unmanaged native modules that execute in the regular user-mode environment. The same user-mode process can host both managed and unmanaged modules interoperating with each other, as will be explained shortly in this section.

Programs in .NET are not compiled directly into native assembly code, but rather into a platform-agnostic language called the *Microsoft .NET Intermediate Language* (usually referred to as MSIL, or simply IL). This IL is then lazily (methods are compiled on first use) turned into assembly instructions by a component of the execution engine called the *Just-in-Time .NET compiler*, or *JIT*.

.NET Side-by-Side Versioning

One of the issues that plagued software development in Windows prior to the introduction of the .NET Framework was the fact that new DLL versions sometimes introduced new behaviors, breaking existing software often through no fault of the application developer. There was no standard way to strongly bind applications to the version of the DLLs that they were tested against before they got released. This is known as the "DLL hell" issue. COM made the situation better by at least ensuring binary compatibility: instead of C-style exported DLL functions simply disappearing or altering their signatures from underneath their consumers (resulting in crashes!), COM servers were able to clearly version their interfaces, allowing COM clients to query the interfaces that they were tested against and providing the safety of this extra level of indirection.

The .NET Framework takes the idea of strong binding and versioning one step further by ensuring that .NET programs are always run against the version of the CLR that they were compiled to target or, alternatively, the version specified in the application's configuration file. So, new versions of the .NET Framework are installed side by side with older ones instead of replacing them. The exception to this is a small shim DLL called *mscoree.dll* that's installed to the *system32* directory (on 64-bit Windows; a 32-bit version is also installed to the *SysWow64* directory) and that always matches the newest .NET Framework version present on the machine. This works because newer versions of *mscoree.dll* are backward compatible with previous versions of the CLR. For example, if both .NET versions 4.0 and 2.0 are installed on the machine, the *mscoree.dll* module in the *system32* directory will be the one that was installed with the CLR 4.0 release.

.NET Executable Programs Load Sequence

The IL assemblies produced by the various .NET compilers also follow the standard Windows PE (Portable Executable) format and are just special-case native images from the perspective of the OS loader, except they have a marker in their PE header to indicate they are managed code binaries that should be handled by the .NET CLR. When the Windows module loader code in *ntdll.dll* detects the existence of a CLR header in the executable PE image it is about to launch, control is immediately transferred to the native CLR entry point (*mscoree!_CorExeMain*), which then takes care of finding and invoking the managed IL entry point in the image.

Note To support earlier OS versions (more specifically, Windows 98/ME and earlier service packs of Windows 2000 and Windows XP), which were not aware of the CLR header because the .NET Framework hadn't shipped at the time those operating systems were released, managed PE images also had to have a regular native entry point consisting of a very thin stub (*jmp* instruction) that simply invokes the CLR's main entry point method (*mscoree!_CorExeMain*). Fortunately, the Windows releases supported by .NET are now natively capable of loading managed code modules, and this stub is no longer strictly required.

Note that the OS module loader doesn't really know which version of the CLR should be loaded for the managed image. This is the role of the *mscoree.dll* native shim DLL, which determines the correct version of the CLR to load. For CLR v2 programs, for example, the execution engine DLL loaded by *mscoree.dll* is *mscorwks.dll*, while for CLR v4 the execution engine implementation resides inside the *clr.dll* module. Once the CLR execution engine DLL for the target version is loaded, it pretty much takes control and becomes responsible for the runtime execution environment, ensuring type safety and automatic memory management (garbage collection), invoking the JIT compiler to convert IL into native assembly code on-demand, performing security checks, hosting the native implementation for several key threading and run-time services, and so on. The CLR is also responsible for loading the managed assemblies hosting the object types referenced by the application's IL, including the core library that implements some of the most basic managed types (*mscorlib.dll* .NET assembly). Figure 1-10 illustrates these steps.

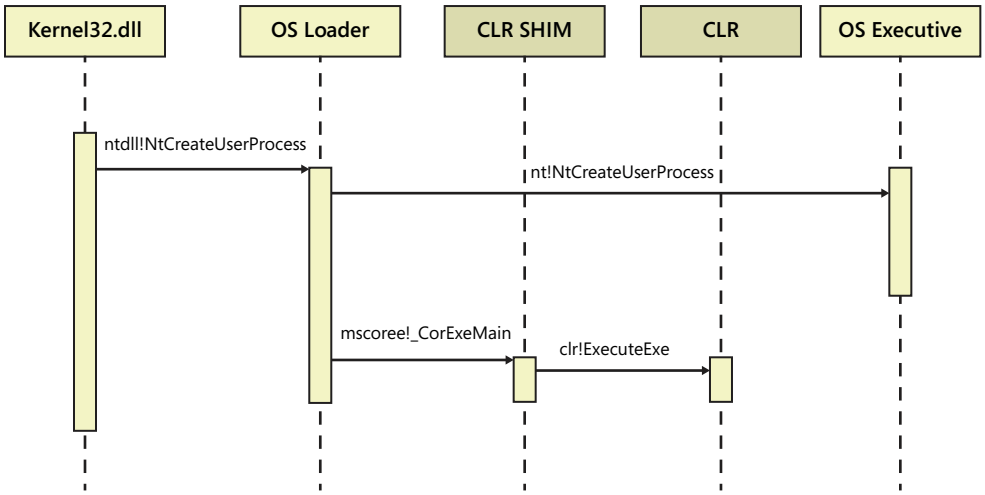


FIGURE 1-10 Loading steps for .NET executable programs.

Interoperability Between Managed and Native Code

When the first version of the .NET Framework was introduced back in early 2002, it had to be able to consume existing native code (both C-style Win32 APIs and COM objects) to ensure that the transition to managed code programming was seamless for Windows developers. Interoperability between managed and unmanaged code is possible fortunately, thanks in large part to two CLR mechanisms: *P/Invoke* and *COM Interop*. P/Invoke (the *DllImport* .NET method attribute) is used to invoke C-style unmanaged APIs such as those from the Win32 API set, and COM Interop (the *ComImport* .NET class attribute) can be used to invoke any existing classic COM object implemented in unmanaged code.

Managed/native code interoperability presents a few technical challenges, however. The biggest challenge is that the garbage collection scheme that the CLR uses for automatic memory management requires it to manage objects in the managed heap so that it is able to periodically clear

defunct (nonrooted) objects, and also so that it can reduce heap fragmentation when dead objects are collected. However, when transitioning to run native code as part of the same process address space, it's often necessary to share managed memory with the native call (in the form of function parameters, for instance). Because the CLR garbage collector is free to move these managed objects around when it decides to perform a garbage collection, and because the garbage collector is completely unaware of the operations that the native code might attempt, it could end up moving the shared managed objects around, causing the native code to access invalid memory. The CLR execution engine takes care of these technical intricacies by marshaling function parameters and pinning managed objects in memory during the managed to unmanaged transitions.

Conversely, you can also consume code that you develop using .NET from your native applications using the COM Interop facilities provided by the CLR. The C/C++ native code is able to consume the types published by a .NET assembly (by means of the *ComVisible* .NET attribute) using their type library, in the same fashion that native COM languages are able to consume types from different languages. The .NET Framework ships with a tool called *regasm.exe*, which can be used to easily generate type libraries for the COM types in a .NET assembly. The .NET Framework and Windows SDKs also include a development tool, called *tlbexp.exe*, that's able to do the same thing.

The CLR shim DLL (*mscoree.dll*) again plays a key role in this reverse COM Interop scenario because it's the first native entry point to the CLR during the COM activation. This shim then loads the right CLR execution engine version, which then loads the managed COM types as they get invoked by the native code. This extends the functionality provided by the COM library in the OS without it having to know about the intricacies of managed code. During the COM activation sequence that the native application initiates, the COM library ends up invoking the standard *DllGetClassObject* method exported from *mscoree.dll*. If you used *regasm.exe* to generate the type library for the C# COM types, *mscoree.dll* also would've been added to the registry as the *InProcServer32* for all the managed COM classes hosted by the .NET DLL assembly. The CLR shim DLL then forwards the call to the CLR execution engine, which takes care of implementing the class factory and standard native COM interfaces on behalf of the managed COM types.

Microsoft Developer Tools

Microsoft typically releases supporting tools (compilers, libraries, and the like) for developers to write code that targets its technologies. These releases are referred to as *development kits*. For example, there is a software development kit (SDK) for developers of Windows Phone applications, a .NET SDK that contains tools to write and sign code for the .NET Framework, an Xbox development kit (XDK) for game developers, and so on. Many of these development kits, including the .NET and Windows Phone SDKs, are available as free downloads from the Microsoft Download Center at *http://www.microsoft.com/downloads*.

The Windows team at Microsoft also ships two important software development kits that include many of the tools presented in this book: more specifically, the Windows Driver Development Kit

(DDK), which contains the build environment used to compile all the native C++ code samples in the book's companion source code, and the Windows Software Development Kit, which contains the Windows debuggers and Windows Performance Toolkit. Both of these development kits are free and available for download from the Microsoft Download Center.

The Windows DDK (WDK)

Each release of Windows is accompanied by a driver development kit targeted for use by Windows driver developers, which contains the headers, libraries, and tools needed for building drivers, as well as several code samples for writing WDM, KMDF, and UMDF drivers.

One of the most useful features included with this kit is a full-blown build and development environment that can be used not only for driver development but for any kind of C/C++ development. It includes C/C++ compilers and many other native development frameworks, including the STL (Standard C++ Template Library) and ATL (Active Template Library for building COM servers) template libraries and their respective implementation headers.

The native C++ code samples from the companion source code use a small portion of ATL that provides support for smart pointers and basic string and collection operations (arrays, hash tables, and so on). Although ATL also comes with the Microsoft Visual Studio suite, the build environment of the DDK was chosen for this book's companion source code so that readers without Visual Studio can follow the case studies and experiments presented in this book.

The Windows SDK

Another important development kit shipped to support new Windows releases is the Windows SDK. Microsoft sometimes ships more than one SDK version per major Windows release: for example, versions 7.0 and 7.1 of the SDK target Windows 7 developers, with version 7.1 bringing many improvements to some of the key tools covered in this book.

The Windows SDK contains useful documentation and samples for building applications on Windows, as well as the public (official) header files and import libraries that are required for compiling your native Windows applications. In addition, the Windows SDK also contains two of the main debugging and tracing tools covered in this book—namely, the Windows debuggers package and Windows Performance Toolkit.

Step-by-step instructions for how to acquire and install those SDK tools will be provided when they're introduced in the following parts of this book so that they're closer to where you will end up needing them.

Summary

This chapter introduced some common terminology and also served as a very short and, admittedly, fast-paced introduction to important layers (Kernel, Executive, NTDLL, Win32, COM/.NET) in the Windows architecture and software development landscape. The following points are also worth remembering as you read the rest of this book:

- Server and client Windows releases share a common kernel and follow a relatively similar release schedule, so it's important to know the client variant of each server release and vice versa. When this book states that a certain kernel feature is added in Windows 7, for example, it's also implied that the Windows Server 2008 R2 kernel has the same capability.
- The CPU architecture of your OS (*x86* or *x64*) is always important to know when performing debugging and tracing experiments.
- When studying a new API set or platform feature, you should try to piece together an architectural diagram in your mind and understand where that new feature fits relative to the existing OS layers and development frameworks. This comes in handy when you later need to debug or trace the code in your software that uses the feature in question.
- When analyzing development frameworks and whether to use them in building your software, you should also try to understand how they are implemented on top of the built-in OS services and what additional functionality they provide. This should help you make an informed choice as to what frameworks fit your scenario best and whether taking the dependency is worth the productivity benefits you would gain.

Many of the concepts discussed in this chapter will be revisited in practical debugging and tracing situations. If you are like me, you will find that these topics start hitting a lot closer to home, so to speak, once you get into the habit of using debugging and tracing to confirm your understanding of the theoretical background. So, don't hesitate to come back and consult this chapter again as you run the debugging and tracing explorations proposed in this book. You might want to double-check some of the theory that has been covered here.

CHAPTER 3

How Windows Debuggers Work

In this chapter

This chapter explains how different types of debuggers work in Microsoft Windows. If you know these architectural foundations, many debugger concepts and behaviors suddenly start making sense. For example, this chapter explains why certain debugger commands and features work only in user-mode or kernel-mode debugging. You'll also dive into the architecture of managed-code debugging and discover why .NET source-level debugging isn't currently supported by the Windows debuggers.

Following that discussion, you'll learn how the architecture of script debugging relates to that of .NET debugging. With HTML5 and JavaScript becoming more prevalent development technologies for rich client user interfaces, script debugging is likely to garner even more attention from Windows developers in the future. This chapter concludes with a section that explains remote debugging and the key concepts that drive its architecture.

User-Mode Debugging

A user-mode debugger gives you the ability to inspect the memory of the target program you're trying to debug, as well as the ability to control its execution. In particular, you want to be able to set breakpoints and step through the target code, one instruction or one source line at a time. These basic requirements drive the design of the native user-mode debugging architecture in Windows.

Architecture Overview

To support controlling the target in user-mode debugging, the Windows operating system (OS) has an architecture based on the following principles:

- When important debug events, such as new module loads and exceptions, occur in the context of a process that's being debugged by a user-mode debugger, the OS generates message notifications on behalf of the target and sends them to the debugger process, giving the debugger program a chance to either handle or ignore those notifications. During each notification, the target process blocks and waits until the debugger is done responding to it before it resumes its execution.
- For this architecture to work, the native debugger process must also implement its end of the handshake, so to speak, and have a dedicated thread to receive and respond to the debug events generated by the target process.
- The interprocess communication between the two user-mode programs is based on a debug port kernel object (owned by the target process), where the target queues up its debug event notifications and waits on the debugger to process them.

This generic interprocess communication model is sufficient to handle all the requirements for controlling the target in a user-mode debugging session, providing the debugger with the capability to respond to code breakpoints or single-step events, as illustrated in Figure 3-1.

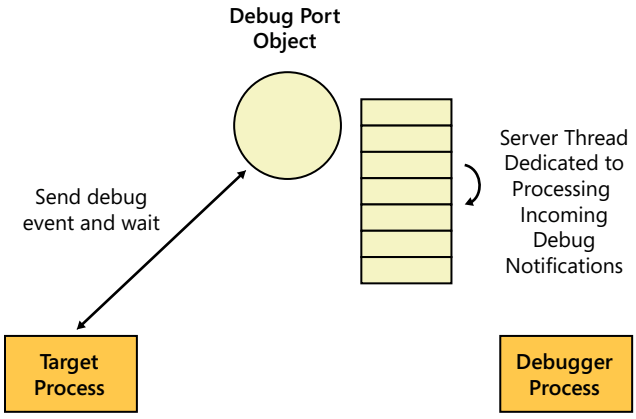


FIGURE 3-1 Native user-mode debugging architecture in Windows.

The other high-level requirement of user-mode debugging is for the debugger to be able to inspect and modify the virtual address space of the target process. This is necessary, for example, to be able to insert code breakpoints or walk the stacks and list the call frames in the threads of execution contained within the target process.

Windows provides facilities exposed at the Win32 API layer to satisfy these requirements, allowing any user-mode process to read and write to the memory of another process—as long as it has

sufficient privileges to do so. This system-brokered access is why you can debug only your own processes unless you're an administrator running in an elevated User Acount Control (UAC) context with full administrative privileges (which include, in particular, the special *SeDebugPrivilege*). If you do have those privileges, you can debug processes from any other user on the system—including *LocalSystem* processes.

Win32 Debugging APIs

Debugger programs can implement their functionality and follow the conceptual model described in the previous section by using published APIs in the operating system. Table 3-1 summarizes the main Win32 functions used in Windows user-mode debuggers to achieve their requirements.

TABLE 3-1 Win32 API Support for User-Mode Windows Debuggers

Requirement	Win32 API Function	WinDbg Command(s)
Start a target process directly under the control of a user-mode debugger.	*CreateProcess*, with *dwCreationFlags*: ■ DEBUG_PROCESS ■ DEBUG_ONLY_THIS_PROCESS	Ctrl+E UI shortcut or windbg.exe target.exe
Dynamically attach a user-mode debugger to an existing process.	*OpenProcess*, with at least the following *dwDesiredAccess* flags: ■ PROCESS_VM_READ ■ PROCESS_VM_WRITE ■ PROCESS_VM_OPERATION *DebugActiveProcess*, with the handle obtained in the previous step	F6 UI shortcut or windbg.exe –pn target.exe or windbg.exe –p [PID]
Stop debugging the target process, but without terminating it.	*DebugActiveProcessStop*	*qd* ("quit and detach")
Break into the debugger to inspect the target.	*DebugBreakProcess*	Ctrl+Break UI shortcut or Debug\Break menu action
Wait for new debug events.	*WaitForDebugEvent*	N/A
Continue the target's execution after a received debug event is processed.	*ContinueDebugEvent*	N/A
Inspect and edit the virtual address space of the target process.	*ReadProcessMemory* *WriteProcessMemory*	Dump memory (*dd*, *db*, and so on) Edit memory (*ed*, *eb*, and so on) Insert code breakpoints (*bp*) Dump a thread's stack trace (*k*, *kP*, *kn*, and so on)

With these Win32 APIs, a user-mode debugger can write the code in the thread that it uses to process the debug events it receives from the target using a loop like the one shown in the following listing (pseudo-code).

```
//
// Main User-Mode Debugger Loop
//
CreateProcess("target.exe",..., DEBUG_PROCESS, ...);
while (1)
```

```
{
    WaitForDebugEvent(&event, ...);
    switch (event)
    {
        case ModuleLoad:
            Handle/Ignore;
            break;
        case TerminateProcess:
            Handle/Ignore;
            break;
        case Exception (code breakpoint, single step, etc...):
            Handle/Ignore;
            break;
    }
    ContinueDebugEvent(...);
}
```

When the debugger loop calls the *WaitForDebugEvent* Win32 API to check whether a new debug event arrived, the call internally transitions to kernel mode to fetch the event from the queue of the debug port object of the target process (the *DebugPort* field of the *nt!_EPROCESS* executive object). If the queue is found to be empty, the call blocks and waits for a new debug event to be posted to the port object. After the event is processed by the debugger, the *ContinueDebugEvent* Win32 API is called to let the target process continue its execution.

Debug Events and Exceptions

The OS generates several types of debug events when a process is being debugged. For instance, an event is generated for every module load, allowing the user-mode debugger to know when a new DLL is mapped into the address space of the target process. Similarly, an event is also raised when a new child process is created by the target process, enabling the user-mode debugging session to also handle the debug events from child processes if it wants to do so.

Debug events are similarly generated when any exceptions are raised in the context of the target process. As you'll shortly see, code breakpoints and single-stepping are both internally implemented by forcing an exception to be raised in the context of the target application, which means that those events can also be handled by the user-mode debugger just like any other debug events. To better understand this type of debug event, a quick overview of exception handling in Windows is in order.

.NET, C++, and SEH Exceptions

Two categories of exceptions exist in Windows: language-level or framework-level exceptions, such as the C++ or .NET exceptions, and OS-level exceptions, also known as *Structured Exception Handling* (SEH) exceptions. Both Microsoft Visual C++ and the .NET Common Language Runtime (CLR) use SEH exceptions internally to implement support for their specific application-level, exception-handling mechanisms.

SEH exceptions, in turn, can be separated into two categories: hardware exceptions are raised in response to a processor interrupt (invalid memory access, integer divide-by-zero, and so on), and software exceptions are triggered by an explicit call to the *RaiseException* Win32 API. Hardware exceptions are particularly important to the functionality of user-mode debuggers in Windows because they're also used to implement breakpoints and in single-stepping the target, two fundamental features of any debugger.

At the Visual C/C++ language level, the *throw* keyword used to throw C++ exceptions is translated by the compiler into a call implemented in the C runtime library, which ultimately invokes the *RaiseException* API. In addition, three keywords (*__try*, *__except*, and *__finally*) are also defined to allow you to take advantage of SEH exceptions and structure (hence the SEH name) your code so that you can establish code blocks to handle or ignore the SEH exceptions that get raised from within that code block. Despite this Visual C++ language support, it's important to realize that SEH is a Windows operating system concept and that you can use it with any language, as long as the compiler has support for it.

Unlike C++ exceptions, which can be raised with any type, SEH exceptions deal with only one type: *unsigned int*. Each SEH exception, whether triggered in hardware or software, is identified in Windows using an integer identifier—the *exception code*—that indicates the type of fault that triggered the exception (divide-by-zero, access violation, and so on). You can find many of the exception codes defined by the OS in the *winnt.h* Software Development Kit (SDK) header file. In addition, applications are also free to define their own custom exception codes, which is precisely what the C++ and .NET runtimes do for their exceptions.

Table 3-2 lists a few common exception codes that you'll see frequently during your debugging investigations.

TABLE 3-2 Common Windows SEH Exceptions and Their Status Codes

Exception Code	Description
STATUS_ACCESS_VIOLATION (0xC0000005)	Invalid memory access
STATUS_INTEGER_DIVIDE_BY_ZERO (0xC0000094)	Arithmetic divide-by-zero operation
STATUS_INTEGER_OVERFLOW (0xC0000095)	Arithmetic integer overflow
STATUS_STACK_OVERFLOW (0xC00000FD)	Stack overflow (running out of stack space)
STATUS_BREAKPOINT (0x80000003)	Raised in response to the debug break CPU interrupt (interrupt #3 on *x86* and *x64*)
STATUS_SINGLE_STEP (0x80000004)	Raised in response to the single-step CPU interrupt (interrupt #1 on *x86* and *x64*)

SEH Exception Handling in the User-Mode Debugger

When an exception occurs in a process that's being debugged, the user-mode debugger gets notified by the OS exception dispatching code in *ntdll.dll* before any user exception handlers defined in the target process are given a chance to respond to the exception. If the debugger chooses not to handle

this *first-chance* exception notification, the exception dispatching sequence proceeds further and the target thread is then given a chance to handle the exception if it wants to do so. If the SEH exception is not handled by the thread in the target process, the debugger is then sent another debug event, called a *second-chance* notification, to inform it that an unhandled exception occurred in the target process.

Figure 3-2 summarizes this OS exception dispatching sequence, specifically when a user-mode debugger is connected to the target process.

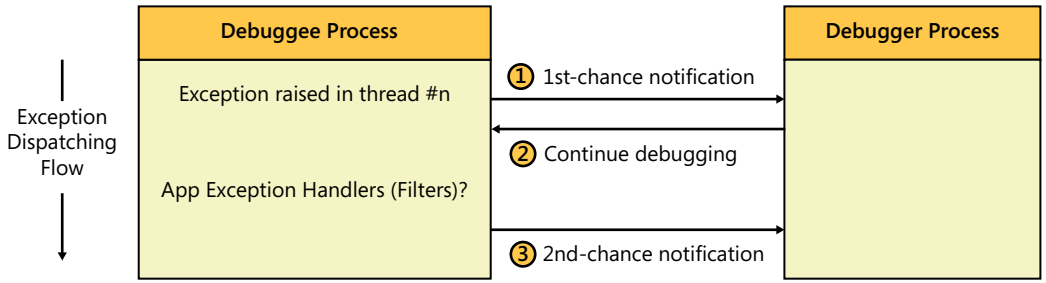


FIGURE 3-2 SEH exceptions and debug event notifications.

First-chance notifications are a good place for the user-mode debugger to handle exceptions that should be invisible to the code in the target process, including code breakpoints, single-step debug events, and the break-in signal. The sections that follow describe these important mechanisms in more detail.

Unlike first-chance notifications, which for user exceptions are simply logged to the debugger command window by default, the user-mode debugger always stops the target in response to a second-chance exception notification. Unhandled exceptions are always reason for concern because they lead to the demise of the target process when no debuggers are attached, which is why the user-mode debugger breaks in when they occur so that you can investigate them. You can see this sequence in action using the following program from the companion source code, which simply throws a C++ exception with a string type. For more details on how to compile the companion source code, refer to the procedure described in the Introduction of this book.

```
//
// C:\book\code\chapter_03\BasicException>main.cpp
//
int
__cdecl
wmain()
{
    throw "This program raised an error";
    return 0;
}
```

When you run this program under the WinDbg user-mode debugger, you see the debugger receive two notifications from the target: the first-chance notification is logged to the debugger command window, while the second-chance notification causes the debugger to break in, as illustrated in the following debugger listing. Notice also how the *throw* keyword used to raise C++ exceptions ends up getting translated into a call to the C runtime library (the *msvcrt!_CxxThrowException* function call in the following listing), which ultimately invokes the *RaiseException* Win32 API to raise an SEH exception with the custom C++ exception code.

```
0:000> vercommand
command line: '"c:\Program Files\Debugging Tools for Windows (x86)\windbg.exe"
c:\book\code\chapter_03\BasicException\objfre_win7_x86\i386\BasicException.exe'
0:000> .symfix
0:000> .reload
0:000> g
(aa8.1fc0): C++ EH exception - code e06d7363 (first chance)
(aa8.1fc0): C++ EH exception - code e06d7363 (!!! second chance !!!)
...
KERNELBASE!RaiseException+0x58:
75dad36f c9              leave
0:000> k
ChildEBP RetAddr
000ffb60 75fd359c KERNELBASE!RaiseException+0x58
000ffb98 00cb1204 msvcrt!_CxxThrowException+0x48
000ffbac 00cb136d BasicException!wmain+0x1b
[c:\book\code\chapter_03\basicexception\main.cpp @ 7]
000ffbf0 76f9ed6c BasicException!__wmainCRTStartup+0x102
000ffbfc 779c377b kernel32!BaseThreadInitThunk+0xe
000ffc3c 779c374e ntdll!__RtlUserThreadStart+0x70
000ffc54 00000000 ntdll!_RtlUserThreadStart+0x1b
0:000> $ Quit the debugging session
0:000> q
```

The Break-in Sequence

User-mode debuggers can intervene at any point in time and freeze the execution of their target process so that it can be inspected by the user—an operation referred to as *breaking into* the debugger. This is achieved by using the *DebugBreakProcess* API, which internally injects a remote thread into the address space of the target process. This "break-in" thread executes a debug break CPU interrupt instruction (*int 3*). In response to this interrupt, an SEH exception is raised by the OS in the context of the break-in thread. As shown in the previous section, this sends the user-mode debugger process a first-chance notification, allowing it to handle this special debug break exception (code *0x80000003*, or STATUS_BREAKPOINT) and finally break in by suspending all the threads in the target process.

This is why the current thread context in the user-mode debugger after a break-in operation will be in this special thread, which isn't a thread you'll recognize as "yours" if you're debugging your own target process. To see this break-in thread in action, start a new instance of *notepad.exe* under the WinDbg user-mode debugger, as shown in the following listing. If you're running this experiment

on 64-bit Windows, you can execute it again exactly as shown here by using the 32-bit version of *notepad.exe* located under the *%windir%\SysWow64* directory on *x64* Windows.

```
0:000> vercommand
command line: '"c:\Program Files\Debugging Tools for Windows (x86)\windbg.exe"  notepad.exe'
0:000> .symfix
0:000> .reload
Reloading current modules.....................
0:000> ~
   0  Id: 1d90.1678 Suspend: 1 Teb: 7ffde000 Unfrozen
0:000> g
```

Using the Debug\Break menu action, break back into the debugger. You'll see that the current thread context is no longer thread #0 (the main UI thread in *notepad.exe*) but rather a new thread. As you can infer from the function name (*ntdll!DbgUiRemoteBreakin*) on the call stack that you obtain by using the *k* command, this is the remote thread that was injected by the debugger into the target address space in response to the break-in request.

```
(1938.1fb0): Break instruction exception - code 80000003 (first chance)
...
ntdll!DbgBreakPoint:
7799410c cc              int     3
0:001> ~
   0  Id: 1d90.1678 Suspend: 1 Teb: 7ffde000 Unfrozen
.  1  Id: 1d90.17f0 Suspend: 1 Teb: 7ffdd000 Unfrozen
0:001> k
ChildEBP RetAddr
00a4fecc 779ef161 ntdll!DbgBreakPoint
00a4fefc 75e9ed6c ntdll!DbgUiRemoteBreakin+0x3c
00a4ff08 779b37f5 kernel32!BaseThreadInitThunk+0xe
00a4ff48 779b37c8 ntdll!__RtlUserThreadStart+0x70
00a4ff60 00000000 ntdll!_RtlUserThreadStart+0x1b
```

In addition, using the *uf* debugger command to disassemble the current function shows that this thread was executing an *int 3* CPU interrupt instruction just before the debugger got sent the first-chance notification for the debug break exception.

```
0:001> uf .
ntdll!DbgBreakPoint:
7799410c cc              int     3
7799410d c3              ret
```

To see the actual threads in the target process, you can use the *~*k* command to list the call stacks for every thread in the target. You can also use the *s* command to change ("switch") the current thread context in the debugger to one of those threads, as illustrated in the following listing.

```
0:001> $ Switch over to thread #0 in the target
0:001> ~0s
```

```
0:000> k
ChildEBP RetAddr
0019f8e8 760fcde0 ntdll!KiFastSystemCallRet
0019f8ec 760fce13 USER32!NtUserGetMessage+0xc
0019f908 0085148a USER32!GetMessageW+0x33
0019f948 008516ec notepad!WinMain+0xe6
0019f9d8 76f9ed6c notepad!_initterm_e+0x1a1
0019f9e4 779c377b kernel32!BaseThreadInitThunk+0xe
0019fa24 779c374e ntdll!__RtlUserThreadStart+0x70
0019fa3c 00000000 ntdll!_RtlUserThreadStart+0x1b
0:001> $ Terminate this debugging session...
0:001> q
```

Setting Code Breakpoints

Code breakpoints are also implemented using the *int 3* instruction. Unlike the break-in case, where the debug break instruction is executed in the context of the remote break-in thread, code breakpoints are implemented by directly overwriting the target memory location where the code breakpoint was requested by the user.

The debugger program keeps track of the initial instructions for each code breakpoint so that it can substitute them in place of the debug break instruction when the breakpoints are hit, and before the user is able to inspect the target inside the debugger. This way, the fact that *int 3* instructions are inserted into the target process to implement code breakpoints is completely hidden from the user debugging the program, as it should be.

This scheme sounds straightforward, but there is a catch: how is the debugger able to insert the *int 3* instruction before the execution of the target process is resumed (using the *g* command) after a breakpoint hit? Surely, the debugger can't simply insert the debug break instruction before the target's execution is resumed because the next instruction to execute is supposed to be the original one from the target and not the *int 3* instruction. The way the debugger solves this dilemma is the same way it is able to support single-stepping, which is by using the TF ("trap flag") bit of the EFLAGS register on *x86* and *x64* processors to force the target thread to execute one instruction at a time. This single-step flag causes the CPU to issue an interrupt (*int 1*) after every instruction it executes. This allows the thread of the breakpoint to execute the original target instruction before the debugger is immediately given a chance to handle the new single-step SEH exception—which it does by restoring the debug break instruction again, as well as by resetting the TF flag so that the CPU single-step mode is disabled again.

Observing Code Breakpoint Insertion in WinDbg

To conclude this section, you'll try a fun experiment in which you'll debug the user-mode WinDbg debugger! Armed with the background information from this section and the familiarity with using WinDbg commands that you've gained so far, you have all the tools to confirm what WinDbg does when a new code breakpoint is added by the user without having to take my word for it.

To start this experiment, run *notepad.exe* under *windbg.exe*. This experiment once again uses the *x86* flavor of *notepad.exe* and *windbg.exe*, but the concepts are identical on *x64* Windows.

```
0:000> vercommand
command line: '"c:\Program Files\Debugging Tools for Windows (x86)\windbg.exe"  notepad.exe'
0:000> .symfix
0:000> .reload
```

Set a breakpoint at *USER32!GetMessageW*, which is a function that you know is going to be hit in response to any user interaction with the *notepad.exe* user interface. Figure 3-3 represents this first debugging session.

```
0:000> bp user32!GetMessageW
```

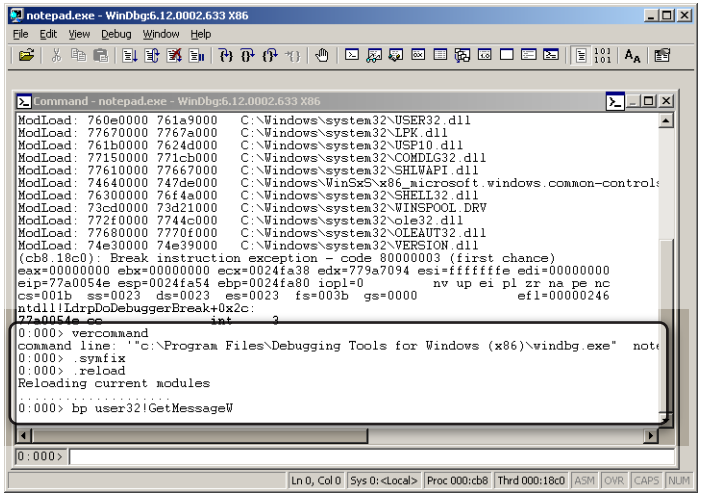


FIGURE 3-3 First WinDbg debugger instance.

Before you use the *g* command to let the target *notepad.exe* continue its execution, start a new *windbg.exe* debugger instance with the same security context as the first one. From this new instance, attach to the first *windbg.exe* process using the F6 shortcut, as illustrated in Figure 3-4. This allows you to follow what happens after you unblock the execution of the *notepad.exe* process from the first debugger instance.

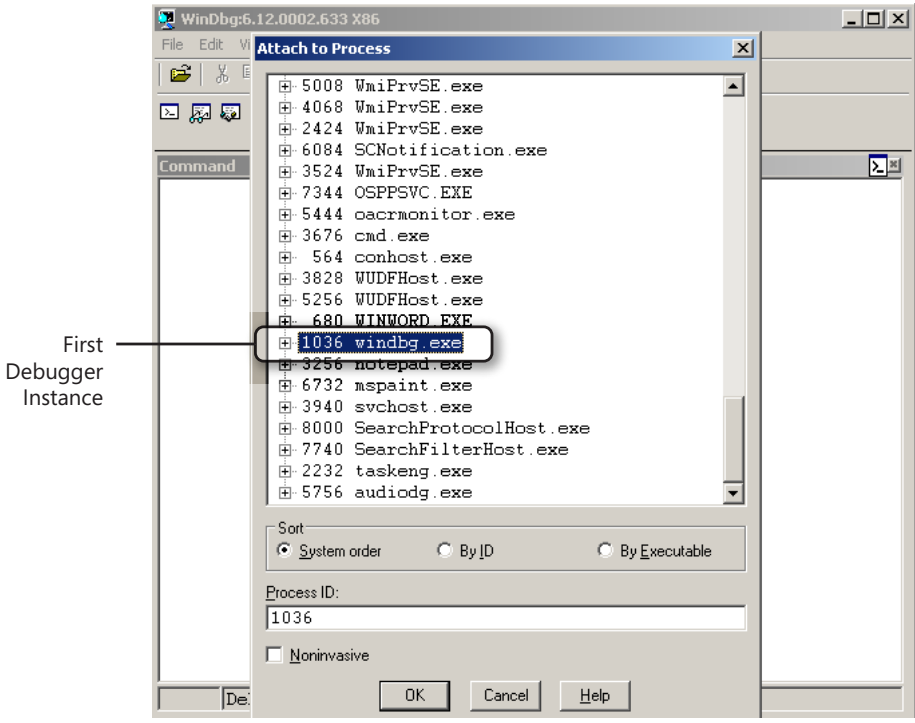


FIGURE 3-4 Debugging the debugger: the second WinDbg debugger instance.

In this new WinDbg instance, set a breakpoint at the *kernel32!WriteProcessMemory* API. As mentioned earlier in this chapter, this is the Win32 API used by user-mode debuggers to edit the virtual memory of their target processes.

```
0:002> $ Second WinDbg Session
0:002> .symfix
0:002> .reload
0:002> x kernel32!*writeprocessmemory*
75e51928 kernel32!_imp__WriteProcessMemory = <no type information>
75e520e3 kernel32!WriteProcessMemory = <no type information>
75eb959f kernel32!WriteProcessMemoryStub = <no type information>
0:002> bp kernel32!WriteProcessMemory
0:002> g
```

Now that you have this breakpoint in place, go back to the first *windbg.exe* instance and run the *g* command to let *notepad.exe* continue its execution.

```
0:000> $ First WinDbg Session
0:000> g
```

Notice that you immediately get a breakpoint hit inside the second *windbg.exe* instance, which is consistent with what you already learned in this chapter, because the first debugger tries to insert an *int 3* instruction into the *notepad.exe* process address space (corresponding to the *USER32!GetMessageW* breakpoint you added earlier).

```
Breakpoint 0 hit
kernel32!WriteProcessMemory:
...
0:001> $ Second WinDbg Session
0:001> k
ChildEBP RetAddr
0092edf4 58b84448 kernel32!WriteProcessMemory
0092ee2c 58adb384 dbgeng!BaseX86MachineInfo::InsertBreakpointInstruction+0x128
0092ee7c 58ad38ee dbgeng!LiveUserTargetInfo::InsertCodeBreakpoint+0x64
0092eeb8 58ad62f7 dbgeng!CodeBreakpoint::Insert+0xae
0092f764 58b67719 dbgeng!InsertBreakpoints+0x8c7
0092f7e8 58b66678 dbgeng!PrepareForExecution+0x5d9
0092f7fc 58afa539 dbgeng!PrepareForWait+0x28
0092f840 58afaa60 dbgeng!RawWaitForEvent+0x19
0092f858 00ebb6cf dbgeng!DebugClient::WaitForEvent+0xb0
0092f874 75e9ed6c windbg!EngineLoop+0x13f
0092f880 779b37f5 kernel32!BaseThreadInitThunk+0xe
0092f8c0 779b37c8 ntdll!__RtlUserThreadStart+0x70
0092f8d8 00000000 ntdll!_RtlUserThreadStart+0x1b
```

You can also check the arguments to the *WriteProcessMemory* API in the previous call stack by applying the technique described in Chapter 2, "Getting Started," where the stack pointer and saved frame pointer values were used to get the arguments to each call from the stack of the current thread. Remember that in the *__stdcall* calling convention, the stack pointer register value points to the return address at the time of the breakpoint, followed by the arguments to the function call. This means that the second DWORD value in the following listing represents the first parameter to the Win32 API call. The values you'll see will be different, but you can apply the same steps described here to derive the function arguments to this API call:

```
0:001> $ Second WinDbg Session
0:001> dd esp
0092edd4  58ce14a2 00000120 7630cde8 58d1b5d8
0092ede4  00000001 0092edf0 00000000 0092ee2c
...
```

In the documentation for the *WriteProcessMemory* Win32 API on the MSDN website at *http://msdn.microsoft.com/*, you'll see that it takes five parameters.

```
BOOL
WINAPI
WriteProcessMemory(
    __in HANDLE hProcess,
    __in LPVOID lpBaseAddress,
    __in_bcount(nSize) LPCVOID lpBuffer,
    __in SIZE_T nSize,
    __out_opt SIZE_T * lpNumberOfBytesWritten
    );
```

The first of these parameters is the user-mode handle for the target process object (*hProcess*) that the debugger is trying to write to. You can use the value you obtained from the *dd* command with the *!handle* debugger extension command to confirm that it was indeed the *notepad.exe* process.

The *!handle* command also gives you the process ID (PID) referenced by the handle, which you can confirm is the PID of *notepad.exe* in the Windows task manager UI or, alternatively, by using the convenient *.tlist* debugger command, as demonstrated in the following listing.

```
0:001> $ Second WinDbg Session
0:001> !handle 120 f
Handle 120
  Type              Process
  GrantedAccess     0x12167b:
         ReadControl,Synch
         Terminate,CreateThread,VMOp,VMRead,VMWrite,DupHandle,SetInfo,QueryInfo
  Object Specific Information
    Process Id  5964
    Parent Process  3736
0:001> .tlist notepad*
 0n5964 notepad.exe
```

Next is the address that the debugger is trying to overwrite (*lpBaseAddress*). Using the *u* debugger command to disassemble the code located at that address, you can see that this second argument does indeed point to the *USER32!GetMessageW* API, which was the target location of the requested code breakpoint.

```
0:001> $ Second WinDbg Session
0:001> u 0x7630cde8
USER32!GetMessageW:
7630cde8 8bff            mov     edi,edi
7630cdea 55              push    ebp
7630cdeb 8bec            mov     ebp,esp
7630cded 8b5510          mov     edx,dword ptr [ebp+10h]
...
```

Finally, the third parameter (*lpBuffer*) is a pointer to the buffer that the debugger is trying to insert into this memory location. This is a single-byte buffer (as indicated by the value of the fourth argument, *nSize*, from the previous listing), representing the *int 3* instruction. On both *x86* and *x64*, this instruction is encoded using the single *0xCC* byte, as you can see by using either the *u* ("un-assemble") or *db* ("dump memory as a sequence of bytes") commands:

```
0:001> $ Second WinDbg Session
0:001> u 58d1b5d8
dbgeng!g_X86Int3:
58d1b5d8 cc              int     3
0:001> db 58d1b5d8
58d1b5d8  cc 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
...
```

If you continue this experiment with the same breakpoints and type **g** again inside the second *windbg.exe* instance, you can similarly analyze the next hit of the *WriteProcessMemory* breakpoint and confirm that the initial byte from the *USER32!GetMessageW* function (*0x8b* in this case) is

surreptitiously restored as the *USER32!GetMessageW* breakpoint gets hit, right before the user is able to issue commands in the first debugger UI, as shown in the following listing.

```
0:001> $ Second WinDbg Session
0:001> g
Breakpoint 0 hit
0:001> k
ChildEBP RetAddr
0092f128 58b84542 kernel32!WriteProcessMemory
0092f160 58adb6a9 dbgeng!BaseX86MachineInfo::RemoveBreakpointInstruction+0xa2
0092f1a4 58ad3bcd dbgeng!LiveUserTargetInfo::RemoveCodeBreakpoint+0x59
0092f1ec 58ad6c0a dbgeng!CodeBreakpoint::Remove+0x11d
...
0:001> dd esp
0092f108  58ce14a2 00000120 7630cde8 00680cc4
0092f118  00000001 0092f124 00479ed8 0000000b
...
0:001> db 00680cc4
00680cc4  8b 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
...
0:001> $ Terminate both debugging sessions now
0:001> q
```

Kernel-Mode Debugging

The high-level requirements for kernel-mode debugging are similar to those of user-mode debugging, including the ability to control the target (break in, single-step, set breakpoints, and so on) and also manipulate its memory address space. The difference in the case of kernel-mode debugging is that the target is the entire system being debugged.

Architecture Overview

Just like in the case of user-mode debugging, the Windows operating system also designed an architecture that answers the system-level needs of kernel debuggers. In the case of user-mode debugging, that support framework is built right into the OS kernel, where the debug port executive object provides the key to the interprocess communication channel between the debugger and target processes. In the case of kernel debugging, the kernel itself is being debugged, so support for the communication channel is built lower in the architectural stack. This is done using Hardware Abstraction Layer (HAL) extensions that implement the low-level transport layer of the communication channel between the host and target machines during kernel debugging.

There are different transport mediums you can use to perform kernel-mode debugging, and each one of them is implemented in its own transport DLL extension. In Windows 7, for example, *kdcom.dll* is used for serial cables, *kd1394.dll* is used for FireWire cables, and *kdusb.dll* is used for USB 2.0 debug cables. These module extensions are loaded by the HAL very early during the boot process, when the target is enabled to support kernel-mode debugging. Because these modules sit very low in the architecture stack, they can't depend on higher-level OS kernel components that might not yet be fully loaded or otherwise turn out to be themselves in the process of being debugged. For that

reason, the KD transport extensions are fairly lightweight and interact directly with the hardware at the lowest possible level without taking any extra device driver dependencies, as demonstrated in Figure 3-5.

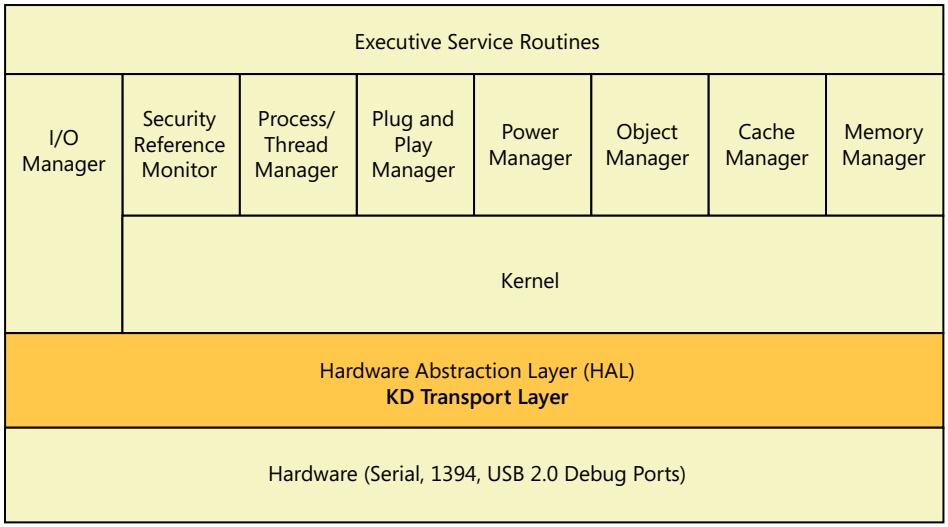


FIGURE 3-5 KD transport layer in the target OS.

If you disregard for a second how the debugger commands are transmitted from the kernel debugger to the target, the conceptual model for how the kernel on the target processes the commands sent by the kernel-mode debugger is quite similar to how debug events are processed by the user-mode debugger loop:

- The OS kernel periodically asks the transport layer (as part of the clock interrupt service routine) to check for break-in packets from the host debugger. When a new one is found, the kernel enters a break-in loop where it waits for additional commands to be received from the host kernel debugger.
- While the system on the target machine is halted, the break-in loop checks for any new commands sent by the host kernel debugger. This enables the kernel debugger to read register values, inspect or change memory on the target, and perform many other inspection and control commands while the target is still frozen. These send/receive handshakes are repeated until the host kernel debugger decides to leave the break-in state and the target is instructed to exit the debugger break-in mode and continue its normal execution again.
- In addition to explicit break-in requests, the kernel can also enter the break-in loop in response to exceptions that get raised by the target machine, which allows the debugger to intervene and respond to them. This generic handling of exceptions is again used to implement single-stepping and setting code breakpoints inside the target OS during kernel-mode debugging.

Setting Code Breakpoints

Knowing how code breakpoints are implemented during kernel-mode debugging is important so that you can understand situations when you fail to hit breakpoints you insert using the host kernel debugger. There are many similarities between how code breakpoints are internally implemented in user-mode and kernel-mode debugging, but there are also several important differences.

Like in the user-mode debugging case, code breakpoints are also inserted by overwriting the target virtual memory address with the debug break CPU instruction (*int 3*). When the target machine hits the inserted breakpoint, a CPU interrupt is raised and its OS interrupt handler is invoked. Where things diverge between user-mode and kernel-mode debugging is in how the handler dispatches the exception event to the host debugger. In the kernel-mode debugging case, the target OS is halted and enters the break-in send/receive loop, allowing the host debugger to handle the breakpoint by putting the initial byte back in the breakpoint's code location before entering the break-in state.

Another way that kernel debugging code breakpoints are different from their user-mode debugging counterparts is that they might refer to memory that has been paged out to disk on the target machine. In that case, the target simply handles the breakpoint command from the host debugger by registering the code breakpoint as being "owed." When the code page is later loaded into memory, the page fault handler (*nt!MmAccessFault*) in the kernel memory manager intervenes and inserts the breakpoint instruction to the global code page at that time, just as it would have done if the breakpoint had been in a memory location that wasn't paged out at the time of the debugger break-in.

Finally, because the same user-mode virtual memory address can point to different private code depending on the user-mode process context, code breakpoints inserted during kernel debugging are always interpreted relative to the current process context. This is a point that sometimes escapes developers who are new to kernel debugging because it isn't a concern in user-mode debugging. However, this is precisely the reason why you should always invasively switch the process context in the host kernel debugger to the target process before setting breakpoints in user-mode code relative to that process.

Single-Stepping the Target

Single-stepping the target in the host debugger is implemented using the same single-step CPU support and interrupt (*int 1*) that enables you to single-step the target process in a user-mode debugging environment. However, the fact that kernel-mode debuggers have global scope again introduces some interesting side effects you should be aware of so that you are better prepared to deal with them during your kernel-debugging experiments.

The most practical difference you'll see when you try single-stepping the target in a host kernel debugger is that execution sometimes seems to jump to other random code on the system and away from your current thread context. This happens when the thread quantum expires while stepping over a function call and the OS decides to schedule another thread on the processor. When that happens, it seems as if the code you're debugging just jumped to a random location. In reality, what happened is that the old thread got switched out and a new one is now running on the processor.

This usually happens whenever you step over a long function or a Win32 API call that causes the thread to enter a wait state (such as a *Sleep* call). Fortunately, when single-stepping in a host kernel debugger, the target OS not only enables the CPU trace flag but cleverly also finds the next call and inserts an additional debug break instruction at that memory location every time you single-step. This means that by letting the target machine "go" again (using the *g* command) after it seemed you had jumped to an unrelated code location, you break right back at the next call from the original thread (once its wait is satisfied and the thread gets scheduled to run again), which allows you to continue single-stepping the thread you were examining prior to the context switch.

Switching the Current Process Context

There are two ways to resolve symbols for the user-mode stacks of a process on the target machine of a kernel-debugging session. The first way, which you already used in Chapter 2, is to simply switch the current process view in the host debugger and reload the user-mode symbols for that process. This method's main advantage is that it also works in live kernel-mode debugging, where it proves useful when you need to observe multiple user-mode processes during a debugger break-in. In the following live kernel-debugging session, the *.process* command is used with the */r* ("reload user-mode symbols") and */p* ("target process") options to illustrate this important approach. Make sure you start a new *notepad.exe* instance and that you use the values in bold text when you execute these commands because your values are likely to be different from the ones shown in this listing.

```
lkd> !process 0 0 notepad.exe
PROCESS 874fa030  SessionId: 1  Cid: 14ac    Peb: 7ffdf000  ParentCid: 1348
    Image: notepad.exe
lkd> .process /r /p 874fa030
Implicit process is now 874fa030
Loading User Symbols.....................
lkd> !process 874fa030 7
PROCESS 874fa030  SessionId: 1  Cid: 14ac    Peb: 7ffdf000  ParentCid: 1348
    Image: notepad.exe
        THREAD 86f6fd48  Cid 14ac.2020  Teb: 7ffde000 Win32Thread: ffb91dd8 WAIT ...
            85685be8  SynchronizationEvent
        ChildEBP RetAddr  Args to Child
        9a99fb10 8287e65d 86f6fd48 807c9308 807c6120 nt!KiSwapContext+0x26
        9a99fb48 8287d4b7 86f6fe08 86f6fd48 85685be8 nt!KiSwapThread+0x266
        9a99fb70 828770cf 86f6fd48 86f6fe08 00000000 nt!KiCommitThreadWait+0x1df
        9a99fbe8 9534959a 85685be8 0000000d 00000001 nt!KeWaitForSingleObject+0x393
        9a99fc44 953493a7 000025ff 00000000 00000001 win32k!xxxRealSleepThread+0x1d7
        9a99fc60 95346414 000025ff 00000000 00000001 win32k!xxxSleepThread+0x2d
        9a99fcb8 95349966 9a99fce8 000025ff 00000000 win32k!xxxRealInternalGetMessage+0x4b2
        9a99fd1c 8283e1fa 000afb80 00000000 00000000 win32k!NtUserGetMessage+0x3f
        9a99fd1c 76f270b4 000afb80 00000000 00000000 nt!KiFastCallEntry+0x12a
        000afb3c 7705cde0 7705ce13 000afb80 00000000 ntdll!KiFastSystemCallRet
        000afb40 7705ce13 000afb80 00000000 00000000 USER32!NtUserGetMessage+0xc
        000afb5c 0055148a 000afb80 00000000 00000000 USER32!GetMessageW+0x33
        000afb9c 005516ec 00550000 00000000 0012237f notepad!WinMain+0xe6
...
```

The second way to switch process views in the host debugger is to perform an *invasive* process context switch on the target machine by using the */i* option of the *.process* command. This method

is particularly useful when you need to set breakpoints in user-mode code locations, given they're always interpreted relative to the current process context on the target machine, as you also learned back in Chapter 2. This method requires the target machine to exit the debugger break-in mode and run to complete the request.

After the target is let go by the host debugger, the kernel on that side thaws the frozen processors and exits the break-in loop. Before it does so, however, it also schedules a high-priority work item to transition over to the new process context that was requested by the host debugger.

```
1: kd> !process 0 0 notepad.exe
PROCESS 874fa030  SessionId: 1  Cid: 14ac    Peb: 7ffdf000  ParentCid: 1348
    Image: notepad.exe
1: kd> .process /i 874fa030
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
1: kd> g
Break instruction exception - code 80000003 (first chance)
```

The work item that induced the previous debug break runs on a *leased* system thread that runs in the context of the requested process. The host debugger breaks right back in again before any of its threads have a chance to continue executing past where they were at the time of the original break-in. You can also confirm that the current thread context is a kernel thread, and not a thread from the user-mode process itself. Notice that thread is indeed owned by the system (kernel) *process*, which always has a PID value of *4*, as reported by the *Cid* (client thread ID) you get from the *!thread* command.

```
0: kd> !thread
THREAD 856e94c0  Cid 0004.0038  Teb: 00000000 Win32Thread: 00000000 RUNNING on processor 0
ChildEBP RetAddr  Args to Child
8a524c0c 82b30124 00000007 8293b2f0 856e94c0 nt!RtlpBreakWithStatusInstruction
8a524d00 8287da6b 00000000 00000000 856e94c0 nt!ExpDebuggerWorker+0x1fa
8a524d50 82a08fda 00000001 a158a474 00000000 nt!ExpWorkerThread+0x10d
8a524d90 828b11d9 8287d95e 00000001 00000000 nt!PspSystemThreadStartup+0x9e
00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x19
```

Nevertheless, this system thread is attached to the target process you requested, which you can confirm using the *!process* kernel debugger extension command and *–1* to indicate you would like the current process context displayed.

```
0: kd> !process -1 0
PROCESS 874fa030  SessionId: 1  Cid: 14ac    Peb: 7ffdf000  ParentCid: 1348
    Image: notepad.exe
```

User-mode code breakpoints you enter in this host debugger break-in state will be resolved relative to this process context, exactly as desired.

Managed-Code Debugging

As previously mentioned in Chapter 2, one of the unfortunate limitations of the Windows debuggers is that they don't support source-level debugging of .NET applications. This doesn't mean that you can't use WinDbg to debug managed code; it simply means that you won't have the convenience of source-level debugging, such as single-stepping and source line breakpoints, when doing so. It's as if you were debugging system code without the source code; only it's worse because many important commands that work for native system debugging, such as displaying call stacks using the *k* command, don't even work for managed code. Fortunately, there is at least a workaround in the form of a WinDbg extension called *SOS*, which Microsoft ships with the .NET Framework. This useful extension is covered in more detail later in this section.

Because of this limitation, the Microsoft Visual Studio environment remains the debugger of choice for .NET debugging. To better understand why the Windows debuggers are lacking in this regard, it's useful to first discuss the architecture used by Visual Studio and the .NET Common Language Runtime (CLR) environment to implement their support for managed-code debugging and understand the way they collaborate to present a seamless native/managed debugging experience. Just like other .NET-related discussions in this book, the coverage centers on the architecture in version 4.0 of the .NET Framework.

Architecture Overview

The first challenge when designing an architecture that enables debugging of Microsoft Intermediate Language (MSIL) .NET code is that such code gets translated into machine instructions on the fly by the CLR's Just-in-Time (JIT) compiler. For performance reasons, this run-time code generation is done lazily only after a method is actually invoked. In particular, this means that to insert a code breakpoint, the debugger needs to wait until the code in question is loaded into memory so that it can edit the code in memory and insert the debug break instruction at the appropriate location. The native debug events generated by the OS aren't sufficient by themselves to support this type of MSIL debugging because only the CLR knows when the .NET methods are compiled or how the managed class objects are represented in memory.

For those reasons, the CLR designed an infrastructure for debuggers to inspect and control managed targets with the help of a dedicated thread that runs as part of every .NET process and has intimate knowledge of its internal CLR data structures. This thread is known as the *debugger runtime controller* thread, and it runs in a continuous loop waiting for messages from the debugger process. Even in the break-in state, the managed target process isn't entirely frozen because this thread must still run to service the debugger commands. Any .NET application will have this extra debugger thread even when it isn't being actively debugged with a managed-code debugger. To confirm this fact, you can use the following "Hello World!" C# sample from the companion source code.

```
C:\book\code\chapter_03\HelloWorld>test.exe
Hello World!
Press any key to continue...
```

You can now use the steps described in Chapter 2 to start a live kernel-debugging session and noninvasively observe the threads that the managed process contains when it's active. Notice the presence of the debugger runtime controller thread (the *clr!DebuggerRCThread::ThreadProc* thread routine in the following listing) even though the .NET process isn't being debugged with a user-mode debugger.

```
lkd> .symfix
lkd> .reload
lkd> !process 0 0 test.exe
PROCESS 85520c88  SessionId: 1  Cid: 07b8    Peb: 7ffdf000  ParentCid: 0e5c
    Image: test.exe
lkd> .process /r /p 85520c88
lkd> !process 85520c88 7
PROCESS 85520c88  SessionId: 1  Cid: 07b8    Peb: 7ffdf000  ParentCid: 0e5c
    Image: test.exe
...
    THREAD 9532ed20  Cid 07b8.1e5c  ...
        828d74b0  SynchronizationEvent
        885f5cb8  SynchronizationEvent
        86a0e808  SynchronizationEvent
...
    0116f7fc 5d6bb4d8 00000003 0116f824 00000000 KERNEL32!WaitForMultipleObjects+0x18
    0116f860 5d6bb416 d6ab8654 00000000 00000000 clr!DebuggerRCThread::MainLoop+0xd9
    0116f890 5d6bb351 d6ab8678 00000000 00000000 clr!DebuggerRCThread::ThreadProc+0xca
    0116f8bc 76f9ed6c 00000000 0116f908 779c377b clr!DebuggerRCThread::ThreadProcStatic+0x83
    0116f8c8 779c377b 00000000 6cb23a74 00000000 KERNEL32!BaseThreadInitThunk+0xe
    0116f908 779c374e 5d6bb30c 00000000 00000000 ntdll!__RtlUserThreadStart+0x70
    0116f920 00000000 5d6bb30c 00000000 00000000 ntdll!_RtlUserThreadStart+0x1b
lkd> q
```

Because of its reliance on this helper thread, the managed-code debugging paradigm is often referred to as *in-process* debugging, in contrast to the out-of-process debugging architecture used by native code user-mode debuggers, which requires no active collaboration from the target process. The contract defined by the CLR for managed-code debuggers to interact with the runtime controller thread is represented by a set of COM interfaces implemented in the *mscordbi.dll* .NET Framework DLL. Because this contract is published as a set of COM interfaces, you can write a managed-code debugger in C/C++, and also in any other .NET language, where the COM Interop facilities can be used to consume the CLR debugging objects implemented in this DLL.

The Visual Studio debugger is based on this same CLR debugging infrastructure, which it also uses to implement its support for managed-code debugging. The components used to service the user actions in the debugger are represented, at a high level, in Figure 3-6. The debugger front-end UI processes any commands entered by the user and forwards them to the debugger's back-end engine, which in turn internally uses the CLR debugging COM objects from *mscordbi.dll* to communicate with the runtime controller thread in the managed target process. These COM objects take care of all the

internal details related to the private interprocess communication channel between the debugger and target processes.

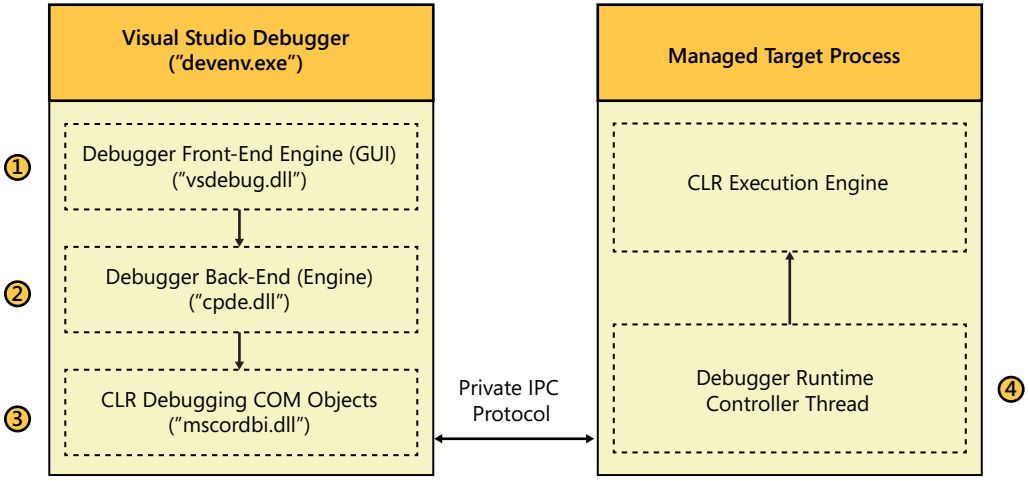


FIGURE 3-6 In-process managed debugging architecture in Visual Studio and the CLR.

This architecture has one big advantage, which is that it insulates the debuggers from the intricate details of the internal CLR execution engine data structures by having a higher-level contract and communication channel between the managed-code debuggers and the CLR debugger controller thread. This means the layouts of those data structures can change without breaking the functionality of those debuggers.

Unfortunately, this architecture also has several drawbacks. First, this model doesn't work for debugging of crash dumps because the target isn't running in that case, so the debuggers can't rely on an active debugger helper thread to perform their actions when debugging a memory crash dump file.

Second, the operating system is unaware that the application is being debugged using this private interprocess communication channel. Up until .NET version 4.0, Visual Studio debugging of managed applications didn't work at all on machines that also had a host kernel debugger attached to them. Because the OS didn't know that the managed process was being debugged, exceptions raised for the purpose of managed debugging were being incorrectly caught by the kernel debugger. The official workaround to this problem was documented in the Knowledge Base (KB) article at *http://support.microsoft.com/kb/303067*, but it's hardly satisfactory because it recommends disabling the kernel debugger entirely. Fortunately, this problem is now fixed in Visual Studio 2010—at least for managed applications compiled for .NET 4.0—because the debugger now also attaches to the target process debug port as a regular native user-mode debugger. However, the in-process managed-debugging architecture is still otherwise being used in that release as the main live, managed-code debugging channel.

Table 3-3 contains a comparative analysis of the in-process and out-of-process debugging architectures.

TABLE 3-3 In-Process and Out-of-Process Managed Debugging Paradigms

	Advantages	Drawbacks
In-process debugging	■ Easy access to CLR data structures ■ Faster single-stepping	■ Poor integration with kernel-mode debugging ■ Doesn't work for crash dump debugging
Out-of-process debugging	■ Supports crash dump debugging ■ Natural integration with native debugging ■ No side effects preventing kernel-mode debugging	■ More difficult for the debugger to stay in-sync with the CLR execution engine's data structures

Given the benefits of out-of-process debugging, the CLR and Visual Studio probably will continue to move toward that architecture for managed-code debugging in the future. That trend has already begun in .NET 4.0 and Visual Studio 2010, where the out-of-process architecture is now used to support crash dump debugging of managed processes.

The SOS Windows Debuggers Extension

Many WinDbg commands don't work natively when debugging a .NET target program. For instance, the *k* command cannot display the names of managed functions in a call stack and the *dv* command cannot display the values of local variables from those functions, either. To understand why, remember that MSIL images are compiled on the fly, so the dynamically generated code addresses are completely unknown to the symbols that the Windows debugger relies on to map the addresses to their friendly symbolic names. Even when an MSIL image is precompiled into a native one—a process known as *NGEN'ing* the assembly—the generated native image is actually machine-specific and won't have a corresponding symbol file, either. The .NET Framework DLL assemblies fall into this second category because they are usually NGEN'ed on the machines where they're installed to improve the performance of all the applications that use them.

How SOS Works

To work around the lack of native support for managed-code debugging in the Windows debuggers, the .NET Framework ships the *sos.dll* debugger extension module. This extension was doubly useful in earlier releases of the .NET Framework because it was also the only supported way to perform crash dump debugging of .NET code, given that Visual Studio started supporting out-of-process debugging of managed code only in its 2010 release.

This debugger extension is built as part of the CLR code base, so it has intimate knowledge of the internal layouts of the CLR data structures, allowing it to read the virtual address space of the target process directly and parse the CLR execution engine structures that it needs. These capabilities

enable it to support out-of-process managed-code debugging. When using SOS, you'll at least be able to display managed call stacks, set breakpoints in managed code, find the values of local variables, dump the arguments to method calls, and perform most of the inspection and control debugging actions that you can use in native-code debugging—only without the convenience of source-level debugging.

Symbols for .NET modules are used by managed-code debuggers only to enable source-level debugging (source lines, names of local function variables, and so on). Even without symbol files for managed assemblies, you still can do a lot of things you aren't able to do in native-code debugging, where the symbols are absolutely crucial. This is because MSIL images also carry metadata describing the type information for the classes they host, allowing any component with internal knowledge of how to parse that information to use it for displaying function names in a call stack, dump the values of local variables (though without their names), or find the parameters to function calls. This is precisely how the SOS Windows debugger extension enables out-of-process managed-code debugging—even without symbol files or any additional help from the CLR debugger runtime controller thread.

Debugging Your First .NET Program Using SOS

To provide a practical illustration for how to use SOS to debug .NET programs in WinDbg, you'll now use it to debug the following C# program from the companion source code, which you should compile to target CLR version 4.0, as described in the procedure provided in the Introduction of this book.

```
//
// C:\book\code\chapter_03\HelloWorld>main.cs
//
public class Test
{
    public static void Main()
    {
        Console.WriteLine("Hello World!");
        Console.WriteLine("Press any key to continue...");
        Console.ReadLine();
        Console.WriteLine("Exiting...");
    }
}
```

Every version of the CLR has its own copy of the SOS extension DLL that understands its internal data structures and is able to decode them. For this reason, you must always load the version of the extension that comes with the CLR version that's used by the target process you're trying to debug. In addition, the SOS commands work only after the CLR execution engine DLL has been loaded, so you need to wait for its module load event to occur. This happens early during the startup of the .NET target as the CLR shim DLL (*mscoree.dll*) hands the reins over to the CLR execution engine DLL, which is *clr.dll* in the case of CLR version 4 (.NET 4.x), and *mscorwks.dll* in the case of CLR version 2

(.NET 2.x and .NET 3.x). You can get notified of this module load event in the debugger by using the *sxe ld* command, as shown in the following listing.

```
0:000> vercommand
command line: '"c:\Program Files\Debugging Tools for Windows (x86)\windbg.exe"
c:\book\code\chapter_03\HelloWorld\test.exe'
0:000> .symfix
0:000> .reload
0:000> sxe ld clr.dll
0:000> g
ModLoad: 5fad0000 6013e000   C:\Windows\Microsoft.NET\Framework\v4.0.30319\clr.dll
ntdll!KiFastSystemCallRet:
779970b4 c3              ret
0:000> .lastevent
Last event: 1e30.c20: Load module C:\Windows\Microsoft.NET\Framework\v4.0.30319\clr.dll at
5fad0000
```

After the execution engine DLL is loaded, you can load the SOS extension module before any managed code has a chance to run inside the target process. A command you'll find useful when loading the SOS extension DLL is the *.loadby* debugger command. This command works just like the more basic *.load* command, but it looks up the extension module under the same path where its second module parameter was loaded from. By specifying the CLR execution engine DLL module name, you will be sure to load the *sos.dll* extension from the same location so that it matches the precise CLR version of the target. One of the useful SOS commands is the *!eeversion* command, which displays the current version of the CLR in the target process.

```
0:000> .loadby sos clr
0:000> !eeversion
4.0.30319.239 retail
0:000> g
```

The program now waits for user input in the *ReadLine* method. If you break into the debugger at this point by using the Debug\Break menu action, you'll see that the *k* command isn't able to properly display the function names in the managed code frames from the main thread in the .NET process. (Notice the very large offsets in the frames from the *mscorlib_ni* native image of the *mscorlib.dll* .NET Framework assembly, which is indicative of missing or unresolved symbols.) The unmanaged frames are still decoded correctly.

```
0:004> ~0s
0:000> k
ChildEBP RetAddr
0017e998 77996464 ntdll!KiFastSystemCallRet
0017e99c 75ea4b6e ntdll!ZwRequestWaitReplyPort+0xc
0017e9bc 75eb2833 KERNEL32!ConsoleClientCallServer+0x88
0017eab8 75efc978 KERNEL32!ReadConsoleInternal+0x1ac
0017eb40 75ebb974 KERNEL32!ReadConsoleA+0x40
0017eb88 5efc1c8b KERNEL32!ReadFileImplementation+0x75
0017ec08 5f637cc8 mscorlib_ni+0x2c1c8b
0017ec30 5f637f60 mscorlib_ni+0x937cc8
0017ec58 5ef78bfb mscorlib_ni+0x937f60
0017ec74 5ef5560a mscorlib_ni+0x278bfb
0017ec94 5f63e6f5 mscorlib_ni+0x25560a
```

```
0017eca4 5f52a7aa mscorlib_ni+0x93e6f5
0017ecb4 5fad21bb mscorlib_ni+0x82a7aa
0017ecc4 5faf4be2 clr!CallDescrWorker+0x33
0017ed40 5faf4d84 clr!CallDescrWorkerWithHandler+0x8e
0017ee7c 5faf4db9 clr!MethodDesc::CallDescr+0x194
0017ee98 5faf4dd9 clr!MethodDesc::CallTargetWorker+0x21
0017eeb0 5fc273c2 clr!MethodDescCallSite::Call_RetArgSlot+0x1c
0017f014 5fc274d0 clr!ClassLoader::RunMain+0x24c
0017f27c 5fc272e4 clr!Assembly::ExecuteMainMethod+0xc1
0017f760 5fc276d9 clr!SystemDomain::ExecuteMainMethod+0x4ec
0017f7b4 5fc275da clr!ExecuteEXE+0x58
...
```

Fortunately, the *!clrstack* command from the SOS debugger extension allows you to see the managed frames in the thread's call stack.

```
0:000> !clrstack
OS Thread Id: 0xe48 (0)
Child SP IP       Call Site
0017eba8 779970b4 [InlinedCallFrame: 0017eba8]
0017eba4 5efc1c8b DomainNeutralILStubClass.IL_STUB_PInvoke(Microsoft.Win32.SafeHandles.
SafeFileHandle, Byte*, Int32, Int32 ByRef, IntPtr)
0017eba8 5f637cc8 [InlinedCallFrame: 0017eba8] System.IO.__ConsoleStream.ReadFile(Microsoft.
Win32.SafeHandles.SafeFileHandle, Byte*, Int32, Int32 ByRef, IntPtr)
0017ec1c 5f637cc8 System.IO.__ConsoleStream.ReadFileNative(Microsoft.Win32.SafeHandles.
SafeFileHandle, Byte[], Int32, Int32, Int32, Int32 ByRef)
0017ec48 5f637f60 System.IO.__ConsoleStream.Read(Byte[], Int32, Int32)
0017ec68 5ef78bfb System.IO.StreamReader.ReadBuffer()
0017ec7c 5ef5560a System.IO.StreamReader.ReadLine()
0017ec9c 5f63e6f5 System.IO.TextReader+SyncTextReader.ReadLine()
0017ecac 5f52a7aa System.Console.ReadLine()
0017ecb4 0043009f Test.Main() [c:\book\code\chapter_03\HelloWorld\main.cs @ 9]
0017eee4 5fad21bb [GCFrame: 0017eee4]
```

The *mscorlib_ni.dll* module shown in the stack trace output of the *k* command is the NGEN image ("ni") corresponding to the *mscorlib.dll* MSIL image. You can treat these modules just like their MSIL sources for the purpose of SOS debugging. In particular, you can set breakpoints at managed code functions from both MSIL or NGEN images by using the *!bpmd* SOS extension command.

For example, you can set a breakpoint at the *WriteLine* method that would be executed by the next line of source code. This .NET method is defined in the *System.Console* class of the *mscorlib.dll* .NET assembly (or in this case, its *mscorlib_ni.dll* NGEN version). The *!bpmd* command takes the target module name as its first argument (without the extension!) and the fully qualified name of the .NET method as its second argument, as shown in the following listing.

```
0:004> !bpmd mscorlib_ni System.Console.WriteLine
Found 19 methods in module 5ed01000...
MethodDesc = 5ed885a4
Setting breakpoint: bp 5EFAD4FC [System.Console.WriteLine()]
MethodDesc = 5ed885b0
Setting breakpoint: bp 5F52A770 [System.Console.WriteLine(Boolean)]
MethodDesc = 5ed885bc
...
```

```
Adding pending breakpoints...
0:004> g
```

This command adds breakpoints to all overloads of the *WriteLine* method (19 of them in the previous case). If you now press Enter in the active command prompt window from the target process, you'll notice that the debugger hits your breakpoint next.

```
Breakpoint 13 hit
mscorlib_ni+0x2570ac:
5ef570ac 55              push    ebp
```

You can again use the *!clrstack* command to see the current stack trace at the time of this breakpoint. The *–a* option of this command also allows you to view the arguments to the managed frames on the stack.

```
0:000> !clrstack -a
OS Thread Id: 0x18e0 (0)
Child SP IP       Call Site
0018f260 5ef570ac System.Console.WriteLine(System.String)
    PARAMETERS:
        value (<CLR reg>) = 0x01fdb24c
0018f264 004600ab Test.Main()
*** WARNING: Unable to verify checksum for test.exe
 [c:\book\code\chapter_03\HelloWorld\main.cs @ 10]
0018f490 5fad21bb [GCFrame: 0018f490]
```

Notice how this command also displays the address of the .NET string object that was passed to the *WriteLine* method, which you can dump using the *!do* ("dump object") SOS debugger extension command.

```
0:000> !do 0x01fdb24c
Name:        System.String
MethodTable: 5f01f92c
EEClass:     5ed58ba0
Size:        34(0x22) bytes
String:      Exiting...
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
5f0228f8  4000103        4         System.Int32  1 instance       10 m_stringLength
5f021d48  4000104        8          System.Char  1 instance       45 m_firstChar
5f01f92c  4000105        8        System.String  0   shared   static Empty
    >> Domain:Value  002a1270:01fd1228 <<
```

Notice that the *!clrstack* command doesn't display the unmanaged functions on the call stack, though it's usually easy to see where the managed calls fit in the overall stack trace by combining the *!clrstack* and the regular *k* back-trace command, which should give you everything you need to know about what code the current thread is currently executing. Note that SOS also has a *!dumpstack* command that attempts to do this merge, but its output can be rather noisy.

The SOS extension also has several other useful commands that you can use to inspect .NET programs, including a variant of the *u* ("un-assemble") command that's also able to decode the addresses of managed function calls in addition to unmanaged addresses. For example, you could use

this command to obtain the disassembly of the current function at the time of the breakpoint in the previous case (the *WriteLine* method).

```
0:000> !u .
preJIT generated code
System.Console.WriteLine(System.String)
Begin 5ef570ac, size 1a
>>> 5ef570ac 55              push    ebp
5ef570ad 8bec            mov     ebp,esp
5ef570af 56              push    esi
5ef570b0 8bf1            mov     esi,ecx
5ef570b2 e819000000      call    mscorlib_ni+0x2570d0 (5ef570d0) (System.Console.get_Out(),
mdToken: 060008fd)
...
```

Notice how the regular *u* command, by contrast, doesn't display the friendly name of the function itself or of the call to *get_Out* (a managed method too) that's made inside the same function.

```
0:000> u .
mscorlib_ni+0x2570ac:
5ef570ac 55              push    ebp
5ef570ad 8bec            mov     ebp,esp
5ef570af 56              push    esi
5ef570b0 8bf1            mov     esi,ecx
5ef570b2 e819000000      call    mscorlib_ni+0x2570d0 (5ef570d0)
```

If you would like to experiment with more SOS debugger commands, you can find a listing of those commands and a brief summary of what they do by using the *!help* command in the WinDbg debugger.

```
0:000> !help
-------------------------------------------------------------------------------
SOS is a debugger extension DLL designed to aid in the debugging of managed
programs. Functions are listed by category, then roughly in order of
importance. Shortcut names for popular functions are listed in parenthesis.
Type "!help <functionname>" for detailed info on that function.
...
0:000> $ Terminate this debugging session now...
0:000> q
```

Table 3-4 recaps the basic SOS commands introduced during this experiment.

TABLE 3-4 Basic SOS Extension Commands

Command	Purpose
!eeversion	Display the target CLR (execution engine) version.
!bpmd	Set a breakpoint using a managed .NET method.
!do (or *!dumpobj*)	Dump the fields of a managed object.
!clrstack *!clrstack –a*	Display the managed frames in the current thread's call stack. The optional *–a* option is used to also display the arguments to the functions on the call stack. These values are the extension's best guess, however; so, they're not always accurate.
!u	Display the disassembly of a managed function.

Despite the fact you can achieve a lot of critical debugging tasks using the SOS extension, the managed-code debugging experience in the Windows debuggers still leaves a lot to be desired. The Windows debuggers clearly are not your first choice when debugging the managed code you write yourself, but SOS can still be a good option, especially if you can't get Visual Studio installed on the target machine or if you are debugging without source code—in which case, you don't lose much by using WinDbg anyway.

Script Debugging

Visual Studio also supports source-level debugging of script languages such as VBScript or JScript. This is internally implemented using the same in-process paradigm that managed-code debugging relies on. One of the reasons that the CLR used the in-process debugging model when it was first released to the public in 2002 was that script debugging had been successfully using it for years (since the mid-90s). In both cases, the debugger needs the script host's or the CLR execution engine's collaboration to support source-level debugging of the target process.

Architecture Overview

To understand how script debugging works, it's useful to first explain a few basic concepts about how script languages are executed in Windows. The key to that architecture is the *Active Scripting* specification. This specification was introduced by Microsoft in the 90s and defines a set of COM interfaces to allow script languages that implement them to be hosted in any conforming host application. In Windows, both VBScript and JScript (Microsoft's implementation of JavaScript) are Active Scripting languages whose implementation fully conforms with that specification.

The Active Scripting specification defines a language-processing *engine*, with the Active Scripting *host* using that engine when the script needs to be interpreted. Examples of Active Scripting engines are *vbscript.dll* and *jscript.dll*, which both ship with Windows under the *system32* directory. Examples of Active Scripting hosts include the Internet Information Services (IIS) web server (server-side scripts embedded in ASP or ASP.NET pages), Internet Explorer (client-side script hosting in web pages), and the Windows scripting hosts (*cscript.exe* or *wscript.exe*) that ship with Windows and can be used to host scripts executed from a command prompt. There are also third-party Active Scripting engines to support other script languages, including Perl and Python.

In addition, the Active Scripting specification also defines a contract (a set of COM interfaces, again) for debuggers to take advantage of the host in their operations. An Active Scripting host that supports debugging (that is, implements the required COM interfaces) is called a *smart* host. All recent versions of Internet Explorer, IIS, and the Windows scripting hosts are smart hosts that implement those interfaces, which is at the heart of the magic that enables Visual Studio to debug scripts hosted by any of those processes. A *Process Debug Manager* (PDM) component (*pdm.dll*) is shipped with the Visual Studio debugger to insulate script engines from having to understand the intricacies of script debugging. In many ways, the PDM component serves the same purpose that the CLR runtime debugger controller thread and *mscordbi.dll* serve during managed debugging, as illustrated in Figure 3-7.

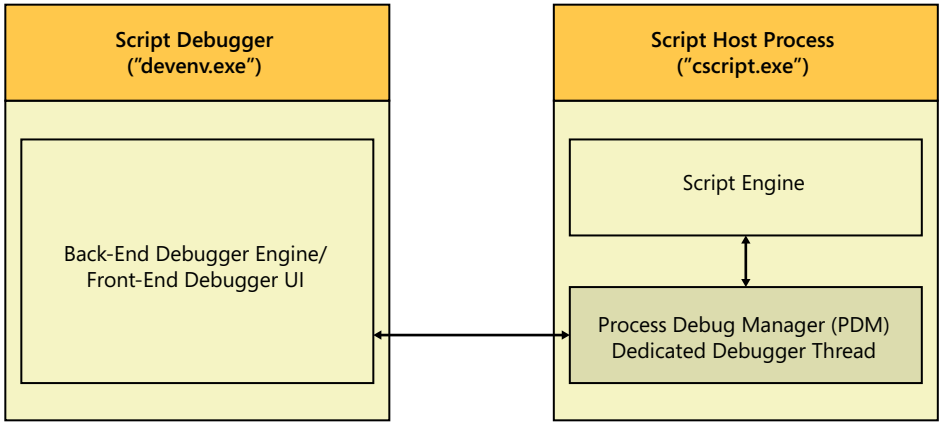


FIGURE 3-7 In-process script debugging architecture.

One way that Active Scripting debugging differs from managed-code debugging is that smart hosts usually do not expose their debugging services by default, whereas in the case of the CLR there is always a debugger thread running in the managed-code process. In Internet Explorer, for instance, you need to first enable script debugging in the host process by clearing the Disable Script Debugging option on the Tools\Internet Options\Advanced tab, as shown in Figure 3-8.

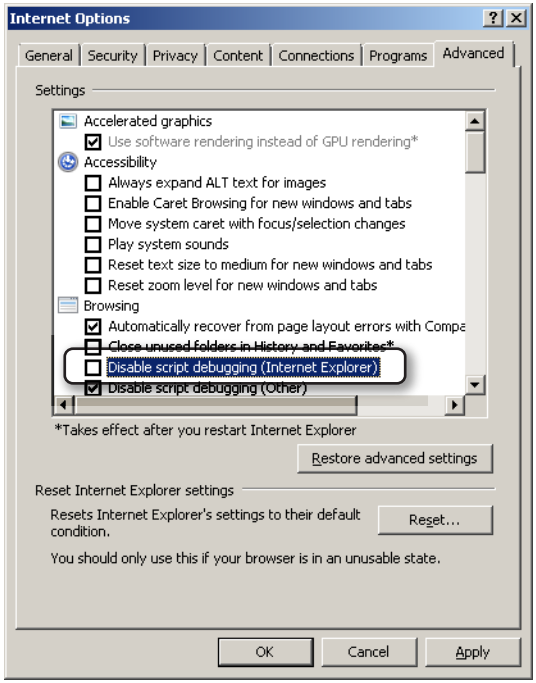


FIGURE 3-8 Enabling Internet Explorer script debugging.

In the same way, you need to explicitly enable debugging of scripts in the Windows scripting hosts (*cscript.exe*/*wscript.exe*) using the *//X* option if you want to also debug the executed scripts. The IIS web server manager also has a UI option for enabling server-side script debugging.

Debugging Scripts in Visual Studio

The following sample script from the companion source code, which simply displays its start time and arguments list to the console before exiting, will serve as a good example for how to use the *//X* option to enable script-debugging support in the *cscript.exe* host process and step through console scripts in Visual Studio.

```
//
// C:\book\code\chapter_03\script>test.js
//
var g_argv = new Array();

//
// store command-line parameters and call the main function
//
for (var i=0; i < WScript.Arguments.length; i++)
{
    g_argv.push(WScript.Arguments(i));
}

WScript.Quit(main(g_argv));

function main(argv)
{
    WScript.Echo("Script started at " + new Date());
    if (g_argv.length > 0)
    {
        WScript.Echo("Arguments: " + g_argv);
    }
}
```

Notice in particular the double slash in the *//X* option in the following command, which *cscript.exe* and *wscript.exe* use to distinguish their own options from the executed script's options.

```
C:\Windows\system32\cscript.exe //X C:\book\code\chapter_03\script\test.js 1 2
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.
...
```

When you run the preceding command on a machine with Visual Studio 2010 installed, you'll be presented with a Visual Studio "attach" dialog box similar to the one shown in Figure 3-9. You might get another dialog box to consent to UAC elevation if you invoked the script from an elevated administrative command prompt, given that the Visual Studio debugger also needs to run elevated in that case.

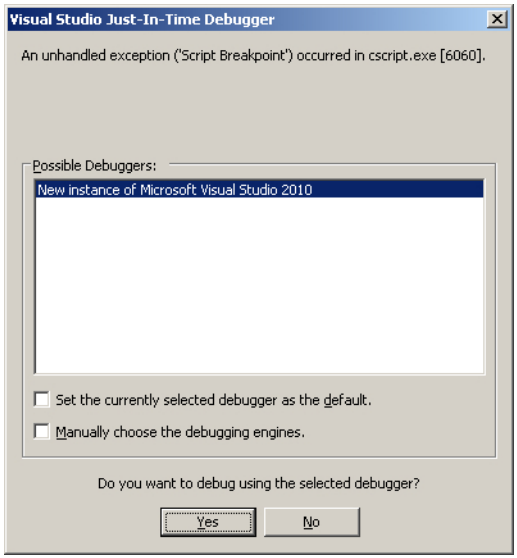


FIGURE 3-9 Attaching to a script using the Visual Studio debugger.

You are then able to use the Visual Studio debugger to step through (using the F10 and F11 shortcuts) the script and debug it (with source-level information!), just as you would with any native-code or managed-code application, as shown in Figure 3-10.

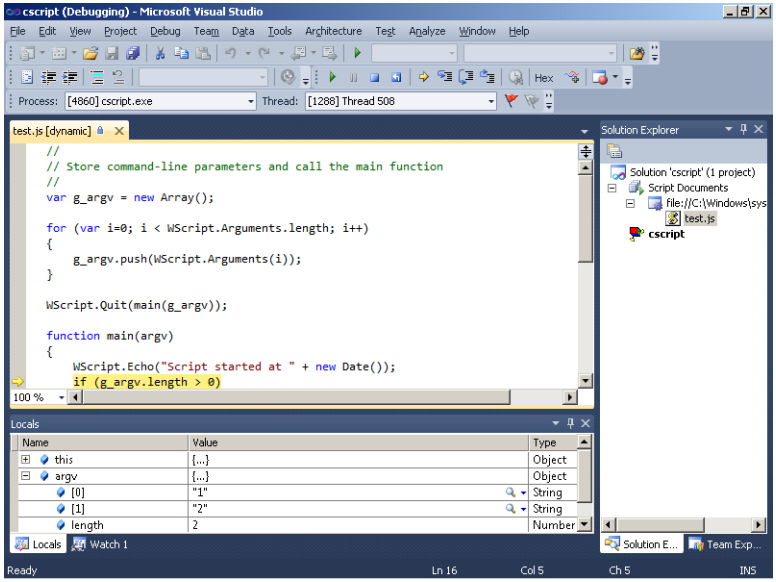


FIGURE 3-10 Source-level script debugging using Visual Studio 2010.

Remote Debugging

Remote debugging is a convenient feature that allows you to control a target remotely using a debugger instance that's running on a different machine on the network. This section provides an inside look at how this feature is implemented by debugger programs in Windows, as well as how you can use it in the case of the WinDbg and Visual Studio debuggers.

Architecture Overview

At a high level, two conceptual models are used to support remote debugging: remote *sessions* and remote *stubs*. In both cases, a process needs to be running on the same machine as the target so that it can carry out the commands that are typed in the remote debugger. In the case of a remote session, the debugger session is entirely on the target machine and the remote debugger instance simply acts as a "dumb" client to send commands to the local debugger instance. In the case of a remote stub, the debugger session is running remotely, with a "stub" broker process running locally on the target machine and acting as a gateway to get information in and out of the target.

Remote sessions are used when collaboration among multiple engineers is needed to investigate a certain failure. In that case, a local debugger instance runs on the repro machine, and developers are then able to start typing commands remotely from their respective machines, take a look at the failure, and even leave comments and see each other's debugging attempts and commands as they get typed in. WinDbg supports this very useful form of remote debugging, which is illustrated in Figure 3-11.

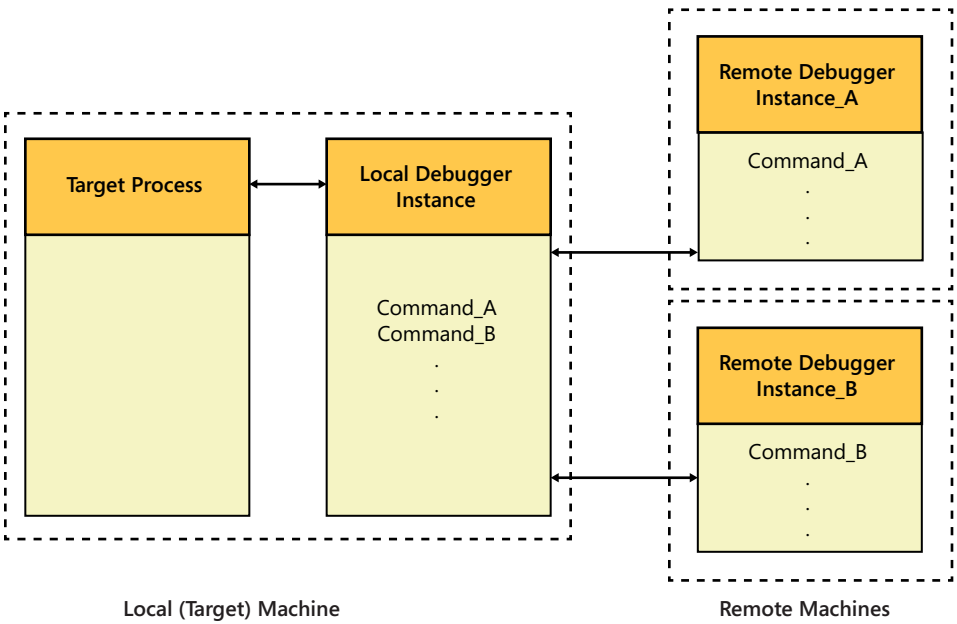


FIGURE 3-11 Remote debugging using remote sessions.

Remote stubs are used when it's important to set up the debugging environment on a remote computer, either because the full debugging environment can't be installed or because the symbols and sources cannot be accessed directly on the target computer. Both the Windows and Visual Studio debuggers support this form of remote debugging, though this architecture is more useful in the case of the Visual Studio debugger. (In fact, remote stubs are the only type of remote debugging that's supported by Visual Studio.) This is because a full Visual Studio debugging environment installation on the target machine is heavy handed, both in terms of the disk space it requires and the time it takes to complete, often making it an inadequate option for production environments. By comparison, the Visual Studio remote-debugging component, which also includes the remote stub process, is more lightweight and can be installed much faster when you need it on the target machine. Figure 3-12 illustrates this second form of remote debugging.

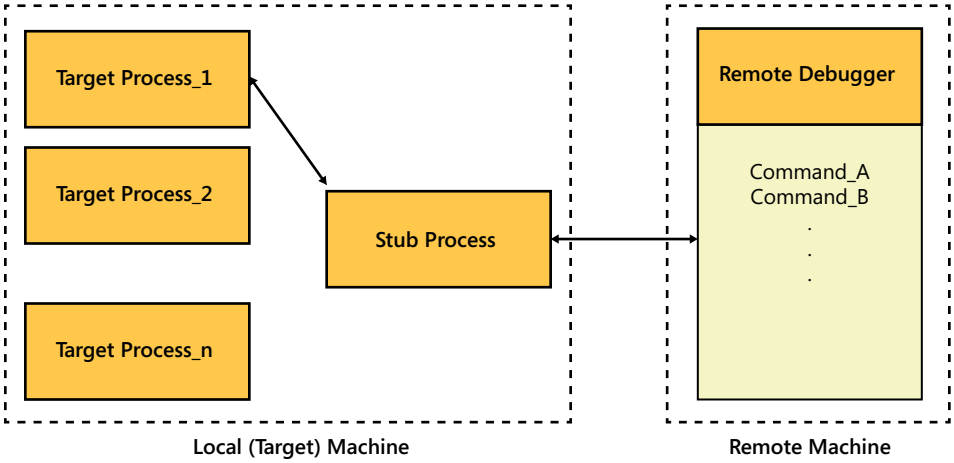


FIGURE 3-12 Remote debugging using remote stubs.

Remote Debugging in WinDbg

As mentioned earlier, WinDbg supports both remote sessions and remote stubs. This section will walk you through the steps for making use of either setup in your remote WinDbg debugging experiments.

Remote Sessions

Starting a WinDbg remote debugging session is straightforward. You use the *.server* debugger command, as illustrated by the following procedure.

Using Remote Sessions in WinDbg

1. From the local WinDbg instance controlling the target process, start a TCP/IP remote session by using the *.server* command, as illustrated in the following listing. This example uses port 4445, but any TCP port that's not currently in use would also work. Using the *.server* command with no arguments reminds you of its syntax.

```
0:000> vercommand
command line: '"c:\Program Files\Debugging Tools for Windows (x86)\windbg.exe"
notepad.exe'
0:000> $ .server without any arguments displays the usage...
0:000> .server
Usage: .server tcp:port=<Socket>  OR  .server npipe:pipe=<PipeName>
0:000> $ Open a remote debugging server session using port 4445
0:000> .server tcp:port=4445
Server started.  Client can connect with any of these command lines
0: <debugger> -remote tcp:Port=4445,Server=TARIKS-LP03
```

2. On the remote machine where you plan to control the target, start a new WinDbg instance (the remote debugger) and connect to the previous TCP port remotely by using the *–remote* command-line option. You can also use the connection string provided in the output from the *.server* command in the preceding listing to remind you of the syntax. Note that the remote and target machines can be the same, so you can also execute this step on the same machine as step 1.

```
windbg.exe -remote tcp:Port=4445,Server=TARIKS-LP03
```

   Notice how any commands you type in the new WinDbg instance's command window (the "remote" debugger) also appear in the first WinDbg instance on the target machine (the "local" debugger). The remote debugger is essentially acting only as a terminal for typing commands that get transferred and, ultimately, processed by the local debugger instance on the target machine.

3. You can terminate the entire debugging session (both the remote and local debugger instances) from the remote machine using the *qq* command. By contrast, the *q* command terminates only the remote instance and leaves the local debugger instance intact.

```
0:000> $ Remote debugger command prompt
0:000> qq
```

In addition to their obvious benefits in remote debugging scenarios, WinDbg remote sessions can be useful even when debugging on the same machine. An example is when you start debugging using one of the command-line Windows debuggers (*cdb.exe* or *ntsd.exe*) and later decide to switch to using WinDbg as a front-end UI to that same session.

Remote Stubs

Commands typed in the remote debugger command window in a remote debugging session are executed as if they were typed on the target machine. In particular, this means that debugger extensions are also loaded into the debugger process on the target machine and that symbols need to be accessible from that machine too. Although remote sessions are by far the more common scenario in WinDbg remote debugging, remote stubs can be used if you don't want to copy the symbols or otherwise make them available over the network so that they're accessible from the target machine.

You can use remote stubs in both user-mode and kernel-mode remote debugging with the Windows debuggers. The *dbgsrv.exe* process, which comes as part of the Windows debuggers package, is used as a stub process in remote user-mode debugging. The *kdsrv.exe* process, also included in the Windows debuggers package, is used in remote kernel-mode debugging.

Just as in the remote session case, you can follow the procedure described here by also running the remote debugger on the target machine, in case you don't have two separate machines.

Using Remote Stubs in WinDbg

1. Open a TCP/IP communication port on the target machine using the *dbgsrv.exe* stub process. This command displays a dialog box when it fails, but it won't show any messages on success.

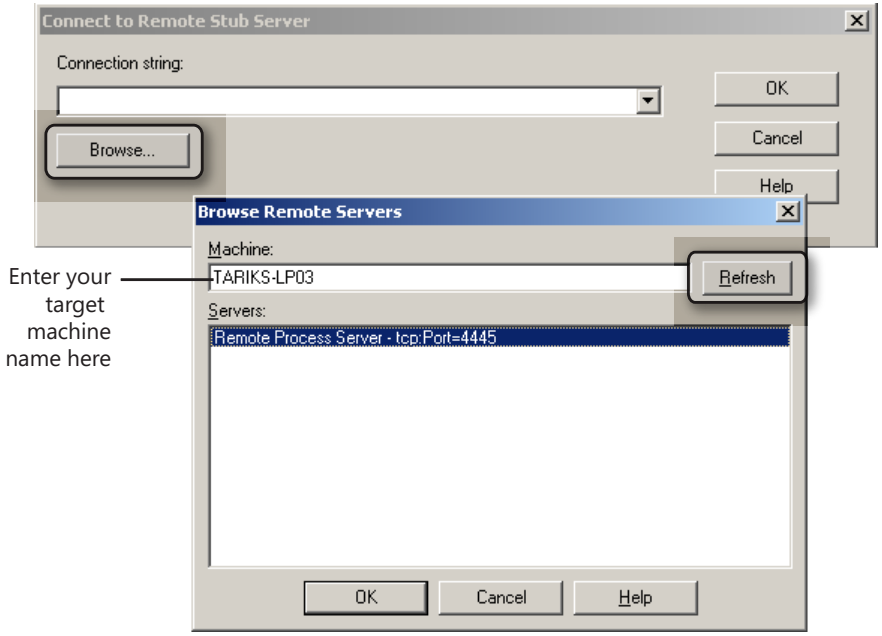
   ```
   C:\Program Files\Debugging Tools for Windows (x86)\dbgsrv.exe -t tcp:port=4445
   ```

   The *dbgsrv.exe* stub process is running in the background at this point. Using the *netstat.exe* tool that comes with Windows under the *system32* directory, you can display the open network ports on the machine and confirm that this stub process is listening for connections from a remote debugger on TCP port 4445:

   ```
   C:\windows\system32\netstat.exe -a
   Active Connections
     Proto  Local Address          Foreign Address        State
   ...
     TCP    0.0.0.0:4445           TARIKS-LP03:0          LISTENING
   ...
   ```

2. On the remote machine, start a new *windbg.exe* instance and connect to the stub process on the target machine using the File\Connect to a Remote Stub menu action. Leave the Connection String combo box empty, and enter the target machine name in the new Browse

dialog box. WinDbg then automatically enumerates all the open stub sessions on the target machine for you, as demonstrated in the following screen shot:



3. After you connect to the remote stub, you can attach to processes running on the target machine by using the familiar File\Attach To A Process menu command (the F6 shortcut). However, this option now shows processes from the target machine, which is exactly what you want in this case. Once attached to a process, you can debug it as if it were running locally, with symbols and debugger extensions located relative to the same remote debugger machine (and not the target machine, as was the case in remote sessions).

4. When you no longer need the remote debugging channel, make sure you terminate the stub process on the target machine so that you release the TCP port it opened earlier for other uses on the machine. You can do that in the Windows task manager UI or by using the *kill.exe* command-line utility from the Windows debuggers package.

```
C:\Program Files\Debugging Tools for Windows (x86)>kill.exe dbgsrv
process dbgsrv.exe (4188) - '' killed
```

This same approach can be used for remote kernel-mode debugging, except you should use the *kdsrv.exe* stub instead of the *dbgsrv.exe* stub. Note that in that case, there are actually three machines involved: the regular target and host kernel debugger machines, and the remote machine you are using to run the debugger instance. The *kdsrv.exe* process is started as a remote stub on the kernel

host debugger machine, not the target machine that is being debugged with the kernel debugger. Symbols and extensions are again resolved relative to the remote debugger machine.

Remote Debugging in Visual Studio

Unfortunately, Visual Studio does not support the concept of remote debugging sessions or remote connection strings, which makes it difficult to use it for sharing a debugging session with another developer at the exact point of a particular failure in the way that WinDbg's *.server* command works. Instead, Visual Studio remote debugging uses the remote stub idea, with the *msvsmon.exe* process acting as the remote stub process to which the Visual Studio debugger process (*devenv.exe*) then connects from the remote machine. MSVSMON is also sometimes called the Visual Studio *debug monitor*, highlighting the fact that it controls the execution of the target on the local machine on behalf of the Visual Studio debugger on the remote machine.

The default communication protocol between the remote and the target machines uses Distributed COM (DCOM) with Windows authentication, so you need to configure the user running the debugger with correct security permissions on the target machine. If the account you're using is a domain user, for example, it will need to be an administrator on the target machine or a member of the *Debugger Users* security group created by Visual Studio.

Using Remote Debugging in Visual Studio

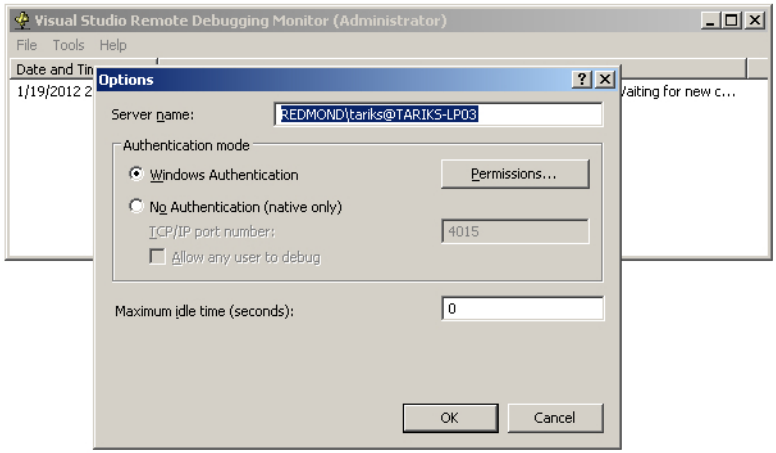
1. Start the following C# sample program from the companion source code (and leave it waiting for user input) on the target machine.

   ```
   C:\book\code\chapter_03\HelloWorld>test.exe
   Hello World!
   Press any key to continue...
   ```

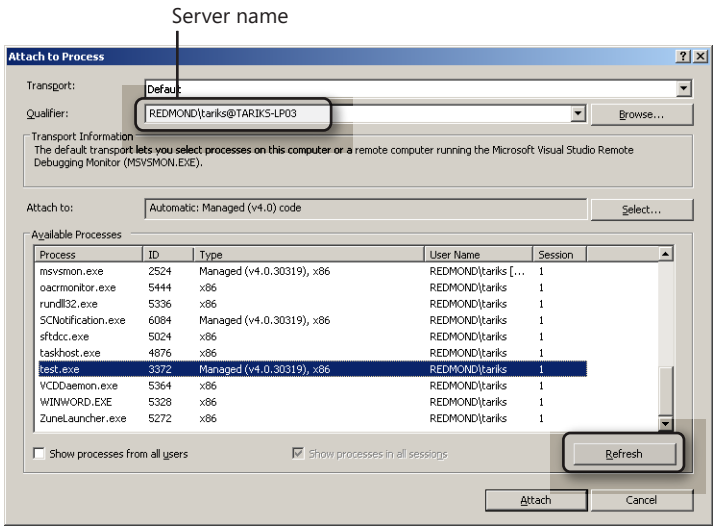
2. Install the Visual Studio 2010 remote-debugging components on the target machine from *http://www.microsoft.com/download/en/details.aspx?id=475*. This setup installation shouldn't take too long (at least compared to the full Visual Studio installation!) because the download is only a few megabytes large. Cancel the configuration wizard that comes up at the end of the installation.

3. Start the *msvsmon.exe* stub process on the target machine. Use the Tools\Options menu action of MSVSMON to change the server name for the connection, or simply leave the default value unchanged, as shown in the following screen shot.

   ```
   C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE\Remote Debugger\x86>msvsmon.exe
   ```



4. On the remote machine, which has the full Visual Studio development environment, use the Tools\Attach To Process menu action to attach to the *test.exe* process remotely. In the Qualifier text box, specify the server name you chose in step 3.



5. If your Windows firewall is enabled (the default behavior), a consent dialog box appears on the target machine. After authorizing the firewall exception, you'll see a list of all the processes that are currently running on the target machine, and you can then attach to the managed *test.exe* process.

You might also see the warning shown in Figure 3-13 as Visual Studio tries to locate the symbols for the managed process. This is because managed-code symbols are resolved relative to the target machine.

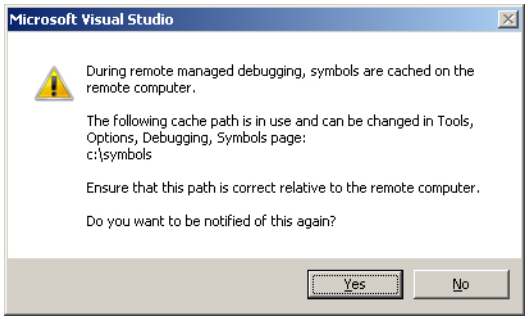


FIGURE 3-13 Managed-code symbols path resolution.

Unlike native code symbols, which are resolved in Visual Studio remote debugging relative to the remote machine (as was the case in the WinDbg stub-based remote debugging case), managed-code symbols are resolved relative to the target machine because of the in-process nature of managed-code debugging. Keep this in mind when you use Visual Studio and you want to perform remote source-level debugging of .NET applications because you need to copy the symbols to the target machine for the debugger to locate them successfully.

Summary

This chapter explained several debugging types and mechanisms in Windows and how they work at the system level. After reading this chapter, the following points should be familiar to you:

- Native user-mode debugging relies on a system-provided architecture that's based on a set of debug events generated by the system on behalf of the target process. These events are sent to a shared debug port object that the OS executive associates with the target process, with a dedicated thread in the debugger process waiting for new events and processing them in that order. This generic framework provides the foundation that user-mode debuggers use to control the target's execution, including setting code breakpoints and single-stepping the target.
- Kernel-mode debugging shares many concepts with user-mode debugging. Because of its global scope, however, a few debugging actions—such as setting code breakpoints and single-stepping the target—are implemented in a slightly different way to accommodate this reality.

- Both script and managed-code debugging use the in-process debugging paradigm, which has a helper thread running in the target process to assist the debugger in its operations. Visual Studio uses this architecture to support both types of debugging.
- The SOS Windows debugger extension can also be used to debug managed code in a completely out-of-process way, though without the convenience of source-level debugging.
- The Windows debuggers support collaborative remote user-mode and kernel-mode debugging using remote sessions. Visual Studio also supports remote debugging, but it uses a different architecture that takes advantage of a remote stub process running on the target machine. The type of remote debugging architecture also affects where symbols and debugger extensions get loaded from, so they each have their practical applications.

Now that you've seen how debuggers can actively debug local and remote targets in Windows, the next chapter will introduce you to another debugging approach when direct live debugging isn't an option. This very important debugging concept is called *postmortem* debugging.

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

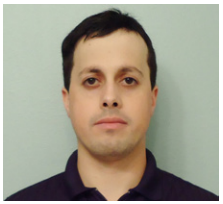
W

X

Y

Z

About the Author



TARIK SOULAMI, a principal development lead on the Windows Fundamentals Team, has more than 10 years of experience designing and developing system-level software at Microsoft. Before joining the Windows team, Tarik spent several years on the Common Language Runtime (CLR) Team, where he helped shape early releases of the Microsoft .NET framework.