

# Win32 Perl Programming

## The Standard Extensions

### Dave Roth

# TABLE OF CONTENTS

ABOUT THE AUTHOR.....	4
<i>About the Technical Reviewers</i> .....	4
<i>Acknowledgments</i> .....	5
TELL US WHAT YOU THINK! .....	5
INTRODUCTION.....	6
<i>Why This Book?</i> .....	6
<i>Who Will Benefit from This Book?</i> .....	8
<i>How This Book Is Structured</i> .....	10
<i>Coding Style</i> .....	12
<i>Hungarian Notation</i> .....	13
<i>Formatting</i> .....	13
<i>Brackets</i> .....	14
<i>Last Thoughts</i> .....	14
CHAPTER 1. WHY PERL ON YOUR WIN32 MACHINE?.....	15
<i>History of Win32 Perl</i> .....	15
<i>ActiveState's ActivePerl</i> .....	16
<i>Perl Modules and Extensions</i> .....	17
<i>Win32 Perl Extensions</i> .....	21
<i>Different Perl Libraries</i> .....	23
<i>Installing Extensions</i> .....	24
<i>Installing Perl</i> .....	28
<i>Parse Exceptions</i> .....	31
<i>Error Handling</i> .....	31
<i>Summary</i> .....	39
CHAPTER 2. NETWORK ADMINISTRATION .....	39
<i>Discovering Servers on the Network</i> .....	40
<i>Finding Domain Controllers</i> .....	47
<i>Resolving DNS Names</i> .....	50
<i>Shared Network Resources</i> .....	56
<i>Managing RAS Servers</i> .....	82
<i>Windows Terminal Server</i> .....	88
<i>Summary: Perl and Win32 Networks</i> .....	105
CHAPTER 3. ADMINISTRATION OF MACHINES.....	106
<i>User Account Management</i> .....	106
<i>Group Account Management</i> .....	124
<i>Removing a Group</i> .....	128
<i>Machine Management</i> .....	133
<i>Summary</i> .....	173
CHAPTER 4. FILE MANAGEMENT.....	173
<i>File Attributes</i> .....	174
<i>File Information</i> .....	179
<i>Shortcuts</i> .....	182
<i>Monitoring Directory Changes</i> .....	196
<i>Summary</i> .....	202
CHAPTER 5. AUTOMATION.....	203
<i>Understanding OLE</i> .....	203
<i>Creating COM Objects</i> .....	208
<i>Interacting with COM Objects</i> .....	217
<i>Destroying COM Objects</i> .....	230
<i>OLE Errors</i> .....	230
<i>Miscellaneous OLE Items</i> .....	231
<i>Summary</i> .....	258
CHAPTER 6. COMMUNICATION.....	259
<i>Sending Messages</i> .....	259
<i>Named Pipes</i> .....	272
<i>Summary</i> .....	288
CHAPTER 7. DATA ACCESS .....	288
<i>What Is ODBC?</i> .....	288

<i>SQL</i> .....	290
USING SQL KEYWORDS .....	291
OLDER VERSIONS OF Win32::ODBC .....	292
<i>Escape Sequences</i> .....	301
<i>How to Use Win32::ODBC</i> .....	305
<i>The Win32::ODBC API</i> .....	314
GETTYPEINFO() AVAILABILITY .....	330
<i>Advanced Features of Win32::ODBC</i> .....	334
<i>Summary</i> .....	343
CHAPTER 8. PROCESSES .....	343
<i>The STD Handles</i> .....	344
<i>Process Management</i> .....	345
<i>Case Study: Running Applications as Another User</i> .....	364
<i>Summary</i> .....	366
CHAPTER 9. CONSOLE, SOUND, AND ADMINISTRATIVE WIN32 EXTENSIONS .....	366
<i>Console Windows</i> .....	366
BUG REPORT .....	374
BUG REPORT .....	382
MAX SIZE AFTER A MANUAL RESIZE .....	385
<i>Sound</i> .....	394
CONSEQUENCES OF PLAYING SOUNDS .....	396
THE VOLUME BUG .....	398
<i>Win32 API</i> .....	399
SPECIFYING A NULL POINTER .....	402
<i>Impersonating a User</i> .....	407
<i>Miscellaneous Win32 Functions</i> .....	409
<i>Summary</i> .....	429
CHAPTER 10. WRITING YOUR OWN EXTENSION .....	429
<i>What Is an Extension?</i> .....	430
<i>What Is a DLL?</i> .....	431
<i>How to Write a Perl Extension</i> .....	431
<i>Beginning Your Extension</i> .....	452
<i>Writing Extension Functions</i> .....	455
PERL v5.006 AND PERL_OBJECT .....	466
<i>A Practical Example</i> .....	471
<i>Summary</i> .....	479
CHAPTER 11. SECURITY .....	479
<i>Security Concepts</i> .....	479
<i>How Does Win32 Security Work?</i> .....	488
LIMITATIONS OF RECURSING WITH WILDCARDS .....	516
<i>Using Win32::FileSecurity</i> .....	521
<i>LSA Functions</i> .....	527
<i>Summary</i> .....	554
CHAPTER 12. COMMON MISTAKES AND TROUBLESHOOTING .....	554
<i>General Win32-Specific Mistakes</i> .....	554
<i>Windows Scripting</i> .....	556
<i>CGI Script Problems</i> .....	563
<i>Win32::NetAdmin</i> .....	570
<i>Win32::Registry</i> .....	571
<i>Win32::ODBC</i> .....	571
<i>Win32::OLE</i> .....	572
<i>Summary</i> .....	573
APPENDIX A. WIN32 PERL RESOURCES .....	574
<i>Book Resources</i> .....	574
<i>Web Resources</i> .....	574
<i>The Official Perl Web Site</i> .....	574
<i>CPAN</i> .....	575
<i>Usenet Resources</i> .....	576
<i>Electronic Magazines and Journals</i> .....	577
<i>Win32 Extensions</i> .....	578

# About the Author



**Dave Roth** is a published writer, columnist, and the author of several popular Win32 Perl extensions including `Win32::AdminMisc`, `Win32::Perms`, `Win32::Daemon` and `Win32::ODBC`. As a leader in the Perl community, Dave has been featured at several conferences, including the O'Reilly Perl conferences and the Usenix LISA-NT conferences. In addition to the first and second editions of this book, he is also the author of *Win32 Perl Scripting: The Administrator's Handbook* (New Riders). Dave also has contributed to *The Perl Journal* and writes a monthly column for the *Windows Scripting Solutions* journal.

Dave has been programming since 1981 using various languages from Assembler to C++ and VB to Perl. His code is used by such organizations as Microsoft, the U.S. Department of Defense, Industrial Light and Magic, Hewlett-Packard, and various colleges and universities. Formerly, Dave helped assemble and administer a statewide WAN for the state of Michigan, and he has designed and administered LANs for Ameritech and Michigan State University.

## About the Technical Reviewers

These reviewers contributed their considerable hands-on expertise to the entire development process for *Win32 Perl Programming: The Standard Extensions*. As the book was being written, these dedicated professionals reviewed all the material for technical content, organization, and flow. Their feedback was critical to ensuring that *Win32 Perl Programming: The Standard Extensions* fits our readers' need for the highest quality technical information.

**Cameron Laird** is vice president of Phaseit, Inc., where he has responsibility for several projects involving both Perl and system administration. Along with everything else that he writes, he maintains the authoritative Perl/Tk FAQ.

**Denis Scherbakov** is one of the system administrators of Belarus National Academy of Science (BNAS). He already has eight years of progressively responsible experience in software development using C/C++, Perl, Java, and management of corporate networks within UNIX, Linux, and Windows. He has held positions that range from designer and software developer to Internet security specialist to system administrator. Denis is currently working on an ozone-aerosol measurement system for lidar stations of the Institute of Physics of BNAS and a European Aerosol Research Lidar Network (EARLINET) to establish an aerosol climatology. He actively cooperates

with international scientific projects such as Aerosol Robotic Network (AERONET) of the NASA Goddard Space Flight Center. Denis lives in Minsk, Belarus, and likes dogs, cats, and his bike.

## Acknowledgments

An unbelievable amount of patience went into this work. My editors, bless their souls, who have weathered through my numerous bouts of mind melt, writer's block, and deadline missing deserve a huge *thank you!* I can only hope that their management reads this and understands that they went beyond the call of duty to make sure this project finished. Management should be told that the delays were not their fault! Big thanks go out to my editors, Karen Wachs, Ann Quinn, and Lisa Thibault. It was their patience, fancy footwork, and empathy that enabled me to finishing this work.

The "patience of Job" award for this book goes to my wife, who gracefully handled another season of "Dave working on his book" again. Even as we discovered the new addition to our family, she continued to be my backbone. My total love and appreciation goes out to Nazli and our yet-to-be-named baby.

Thanks go out to my family and friends who have managed to nag me enough to get the book done. Now that it is all over, we can go hiking again and fly back to Michigan to see the clan!

And finally, a thank you goes out to everyone who provided support and encouragement for the book. The feedback everyone provided has been tremendous. The community needs good resources like this, so keep writing that code and spreading the word about Perl!

## Tell Us What You Think!

As the reader of this book, you are the most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an Executive Editor at New Riders Publishing, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

*Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.*

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax: 317-581-4663

Email: [stephanie.wall@newriders.com](mailto:stephanie.wall@newriders.com)

Mail:  
Stephanie Wall  
Executive Editor  
New Riders Publishing  
201 West 103rd Street  
Indianapolis, IN 46290 USA

# Introduction

Since I published the first edition of *Win32 Perl Programming: The Standard Extensions*, quite a bit has changed. Perl has changed versions, a slew of new extensions have been written, bugs have been squashed, new bugs have been discovered, and the cycle has continued. The book, however, has remained relevant regardless of these changes. This is a testimony to the need for such a book.

My goal was to write a book that was concise, informative, and useful. I didn't want the book to become just another stack of paper sitting on a shelf or propping up someone's computer monitor. I wanted the book to matter, to be something that is kept at an arm's reach from the keyboard. It appears that this was exactly what happened.

In my ideal world, all reference books would be just that, used for reference. I certainly don't have time to read through paragraphs of some author droning on with stories of his youth. Instead, I want the author to tell me what I need to know. After all, if I wanted a biography, I wouldn't have been looking in the computer section of the bookstore.

The goal of this second edition is much like the goal of the first edition. Basically, the book has been updated to reflect the latest version of Perl (version 5.6) and what it is capable of. While researching what to add to the second edition, I was pained in having to decide what to put in and what to leave out. Just how does one decide if one extension is more useful than another, anyhow? The criteria I decided to use were based on the amount of questions asked on several different forums. Email lists, Usenet groups, personal messages people have sent me as well as discussions at various conferences were among the list of sources I used. I also considered what extensions are the basis of others. For example, the `Win32::API` extension is used for `Win32API::Net`, `Win32API::File` and `Win32API::Registry`. So it makes sense to explain `Win32::API`.

I know that including some extensions but excluding others might seem like there is some sort of agenda involved. I can assure you that there is none. I concentrate on the "standard extensions," those that either a) come with Win32 Perl's standard library and are commonly used, or b) are so very commonly used that they deserve recognition (such as `Win32::Lanman`).

## Why This Book?

Microsoft's Win32 platforms (such as Windows 95/98/ME and NT/2000/XP) have become quite popular. The reasons behind this depend on how you view the industry. Some believe that Win32 is so common because it is a good operating system. Yet others believe the sole reason for its popularity is due to its presence on practically every new PC being shipped. Regardless of the reason, it is uncontested that, indeed, more and more people are using the Win32 platform.

Along with this surge of Win32 users comes a surge of Win32 Perl programmers. These coders are unique from most others in the Perl community. They have long been pampered with the Microsoft Windows GUI environment that provides clever and convenient techniques to retrieve information. By using icons, graphics, sounds, and a host of other sense-provoking resources, the capability to access programming documentation is effortless and even enjoyable.

The documentation that exists regarding Perl's Win32 extensions, however, is far from convenient. Typically, it includes a list of functions, the parameters they accept, and a brief description of what

each function does. But for a Perl programmer new to Win32 or someone new to programming in general, such limited documentation can leave a person lost, confused, and frustrated. This becomes self-evident if you peruse the Usenet's Perl forums where newbies post questions that others bemoan as being obvious or in some obscure FAQ.

This book takes aim at this issue and not only documents the Win32 extension's functions but also explains what they do, why they do it, and goes into detail explaining what to expect as a result of the function. For many functions, there are in-depth details that give the reader the knowledge of not only how to use the extensions but also when to use them.

## Lack of Documentation

Because the Perl Win32 extensions generally are written and put out on the Internet without funding, it stands to reason that there would be a lack of documentation. After all, it takes lots of time to create, test, and maintain an extension let alone write comprehensive tutorials. Any documentation that an author provides is just icing on the cake. Having written several extensions myself, I can assure you that it sucks up quite a bit of time maintaining them.

Because an extension is not really useful unless you know how it is used, it is reasonable to feel that documentation is quite important. The problem is that the lack of adequate documentation compounded with the need to study such information creates a knowledge void. This is far more prevalent with the Win32 extensions than it is with Perl in general. You can find a plethora of Web pages rehashing Perl's main pages, but almost all of these are based on the non-Win32-specific extensions. This does not help my clients who are looking for documentation. They quite often ask me what the best resources are for learning how to use the Win32 extensions. My reply is that there are four basic avenues to follow.

## Learning from the Net

The Internet contains a wealth of information on the Win32 extensions. [Chapter 1](#), "Why Perl on Your Win32 Machine?" and [Chapter 12](#), "Common Mistakes and Troubleshooting," list Internet resources that are very valuable to anyone programming the Win32 extensions. Typically, these resources either are very vague and not too helpful, or they are very specific and technical but do not explain how one extension works with another.

## Learning by Reading the Source Code

Studying an extension's source code is the *absolute best way* to learn how it works. This is where you learn the full details on how a particular extension implements its features and even uncovers undocumented features. It also lets you discover bugs that may explain why a function does not work as advertised.

The biggest drawback of this learning technique is that many of the Win32 Perl programmers either are not familiar with C/C++ or don't understand the Perl extension API. For those who feel comfortable with the C programming languages, [Chapter 10](#), "Writing Your Own Extension," explains the details of a Win32 extension.

## Learning by Experimentation

Experimentation is one of the most valuable ways to discover the uses of an extension. By creating test scripts that call extension functions in several different contexts, the treasures hidden within a

Win32 extension can all fall into place. This particular style of learning, however, can be time consuming and can lead to quite a bit of frustration.

## **Learning via Books**

It is books like this that give the reader insight into the extensions. My bookshelves are filled with reference books ranging from Win32 and C++ to AppleTalk and cryptography. A good reference book is more valuable xv than gold to a programmer. It can save hours of calling up friends, accessing Web search engines, and ripping apart header files. It is the aim of this book to find a spot on your bookshelf with other reference books that you find invaluable.

## **What You Can Expect from This Book**

It seems as if most Win32 Perl books that have come out lately try to address so much that the details are lost. It does me no good to have 10 books that all lightly cover the same topics; I would rather have 5 books that each address different topics in detail.

This book neither tries to teach Perl nor explain the Win32 API. Instead, it explains Perl's standard Win32 extensions. Everything from [Chapter 1](#) to Appendix C, "Win32 Network Error Numbers and Their Descriptions," revolves around these extensions: how to use them, when to use them, and how to understand them. Along the way, I explain many aspects of the Win32 API, Perl, and other related topics, but I focus on the extensions. This book was written to compliment good Perl programming and Win32 API books, not replace them.

## **Who Will Benefit from This Book?**

This book is not really for the Perl beginner. Someone just starting to learn the intricacies of Perl might find this resource a bit confusing. This is not to say that beginners should avoid it, but they should understand that this book assumes at least a rudimentary understanding of Perl. Concepts such as regular expressions, scalar variables, hashes, arrays, modules, extensions, and others are routinely tossed around. If a beginning Perl programmer reads this text without a good resource guide to explain these concepts, it will probably be of little use.

Just as a Perl newbie might find this book to be a bit more than he bargained for, so will someone with no Win32 background. This book assumes that the user is familiar with Win32 concepts such as domains, user and group accounts, the Registry, primary and backup domain controllers, networking, and remote access services (RAS).

Keep in mind that lacking an understanding of these concepts does not preclude anyone from reading and appreciating this book. It simply means that the reader might need to consult other resources to understand some of the concepts. Some concepts, however, are described because of the nature of the topic. For example, the topics of COM, OLE, ODBC, and named pipes are directly related to the way particular extensions function. Therefore, a rudimentary description of these concepts is provided to facilitate learning how to use their respective extensions.

So there you have it. This book targets intermediate and advanced Perl programmers who have at least a basic understanding of Win32 platforms. These readers can be very generally divided into five categories (for the sake of simplicity and brevity, I will address only five).

## **Win32 Users**

Users make up the bulk of the Win32 market. Some are forced to use it at work, and others use it at home. Yet others fall into fringe categories but nonetheless use the platform. Many of these people will eventually need to automate processes for a variety of reasons. Maybe they will want all temporary files deleted from the hard drive upon boot up. Maybe some will want to automatically download a Web page every hour. Who knows what would cause a user to want to learn Perl, but indeed, users are learning it.

This book explains how most of the common Perl Win32 extensions work. While relating this information, the book also explains many of the fine points that help clarify why a function might perform in a way that might not seem reasonable. Tips and notes also pave the way for any user who is learning either Win32 Perl or the Win32 platform (or both).

## **Win32 Administrators**

Administrators have the greatest reason to benefit from using Perl. Having been a system administrator for WANs with thousands of users, I can say without any doubt that Perl has saved my hide too many times to count.

From simple utilities such as logon scripts to automatic account-creation scripts, Perl is an administrator's most indispensable tool. My colleagues and I have used Perl scripts as a poor man's Systems Management Service (SMS), allowing me to install, update, manage, and administrate thousands of client and server machines across LANs, WANs, internets, and intranets.

Win32 administrators will find this book particularly important because it covers the most important administration functions. Topics such as user, group, and machine management are covered as well as permissions, sending messages, and file management.

## **Win32 Programmers**

Sure you can program some rather sophisticated applications using C, Pascal, or Visual Basic, but for quick prototyping, I will bet that Perl would beat most any language hands down. The capability to quickly and effectively slap together a working script for a proof-of-concept project is one of Perl's great capabilities. For example, I can write code that connects to an ODBC data source, queries the database, and retrieves and processes the results faster using Perl than I can open Microsoft Access and create a query by dragging and dropping icons.

Any programmer will feel at home with Perl by her side. Even if she uses another language, Perl can still be the reliable tool that saves the day. Recently, I had to walk through all the headers of one of our C++ projects, picking out error messages and their associated error code values. The source tree was 38MB of source code, which equates to quite a few header files. By taking 10 minutes to write a Perl script, I was able to process all header files in under 15 minutes (including the time to write the script)!

Programmers will find this book to be a valuable reference. Because Perl's Win32 extensions make copious use of the Win32 API, any Win32 programmer will have a better understanding of which extension provides the best interface to achieve a desired goal.

## **Win32 Webmasters**

With all the hype about the Internet, Webmasters have found it necessary to learn how to deal with programming issues such as Active Server Pages (ASPs), common gateway interfaces (CGIs), data source names (DSNs), user accounts logging on as services, permissions, and a variety of other issues. Of course, Win32 Perl is a natural in these regards. This book not only addresses common traps that Webmasters fall into, it explains how to avoid them.

## **UNIX Users**

For our UNIX brethren who have found it necessary to migrate data or machines to the Win32 world, this book can be a lifesaver. Perl's Win32 extensions can ease the migration by providing functions that are needed to make a UNIX user's pilgrimage into the Win32 landscape a bit more familiar. This book covers topics such as scheduling batch jobs to run (similar to CRON entries), InterProcess Communication (IPC) such as named pipes, shortcuts (a kind of symbolic links), and permissions, and user and group management.

## **How This Book Is Structured**

The way that this book is structured is a bit of a departure from other books that cover similar topics. It seems that most of them address Perl's Win32 extensions by name. That is to say, each chapter covers a particular extension.

One of the biggest complaints I have received from coders on the Internet relates to not knowing what extension performs which function. Now consider this: If you don't know which extension performs which function, how useful is a book that assumes you know each extension's functionality? If you wanted to create a new user group, would you intuitively know to look up the chapter that covers `Win32::NetAdmin`? Or if you needed to get a list of the CD-ROM drives on your computer, would you know to flip to the pages that discuss the `Win32::AdminMisc` extension? Most users who have talked with me about these issues tell me that they would have no idea.

This book addresses this problem by designating each chapter with a programming topic. These chapters cover a topic that a Win32 Perl programmer might find useful from computer administration and automation to accessing database data and interfacing directly with the Win32 API. There are some extensions, however, that are so specific in their functionality that they really only apply to one topic, so they end up getting a chapter all to themselves (such as the chapters on OLE and ODBC).

The chapters break out into the following discussions:

### **Chapter 1: Why Perl on Your Win32 Machine?**

This chapter discusses the history of the Win32 port of Perl, exactly what extensions are, and how they are used. This includes an examination of the differences between methods and functions. A discussion regarding how to handle errors when using Perl's Win32 extensions is also included.

## **Chapter 2: Network Administration**

[Chapter 2](#) covers the basics of network administration. This includes discussions on discovering the machines on your network, resolving DNS names, managing shared resources, and managing RAS servers.

## **Chapter 3: Administration of Machines**

The details of managing a computer are discussed with an emphasis on user and group accounts, user RAS privileges, INI files, the Registry, and event logs.

## **Chapter 4: File Management**

This chapter is all about files. Any user who is looking for information on how to manage file attributes would want to crack open this chapter. Win32 shortcuts are also covered, including how to create, manage, update, and assign hotkeys to them. Finally, the art of monitoring a directory for changes is explained.

## **Chapter 5: Automation**

The ability to automate programs by using OLE is discussed in [Chapter 5](#). Here you will learn not only what automation is but how it works and how you can use it to get the most out of Windows programs.

## **Chapter 6: Communication**

[Chapter 6](#) covers the details surrounding the Win32 communication techniques of message sending and named pipes.

## **Chapter 7: Data Access**

This chapter dedicates itself to accessing databases using the `Win32::ODBC` extension. What ODBC is and how your script can interact with it are what you can expect. Because SLQ is the query language of choice for ODBC, it is discussed as well.

## **Chapter 8: Processes**

Process management is covered with an emphasis on process creation. You will learn how to spawn a new process using various techniques.

## **Chapter 9: Console, Sound, and Administrative Win32 Extensions**

This chapter covers some of the miscellaneous tidbits that don't seem to find a home in the other chapters. Here you will find a detailed discussion about controlling consoles (a.k.a. Dos boxes), playing sound files (in the `.WAV` format), and interacting with the Win32 API. Additionally, the miscellaneous functions found in the `Win32.pm` module (and extension) are discussed.

## **Chapter 10: Writing Your Own Extension**

Whether you are writing an extension or reading the source code for an existing extension, this chapter explains the details that make it all make sense. Explanations of scalar variables, arrays, hashes, and references are provided. For anyone who has tried to read an extension's source code, this chapter describes the entire process.

## **Chapter 11: Security**

One of the most powerful aspects of Windows NT/2000/XP machines is that they support true security. This enables an administrator to set permissions on particular objects such as files, directories, printers, network shares, and others. However, although this is a powerful tool, it is also very difficult to understand and program. This chapter describes how Win32 security works and how you can use Perl to discover and modify security settings.

## **Chapter 12: Common Mistakes and Troubleshooting**

This chapter describes some of the more common problems that a programmer will run into when using Perl's Win32 extensions. This includes not only extension-specific problems but also CGI and ASP issues.

## **Appendix A: Win32 Perl Resources**

This appendix is a detailed reference that illustrates the all of the functions in the extensions covered in this book. Each function's syntax is shown along with a brief description of what it does.

## **Appendix B: Win32::ODBC Specific Tables**

There have been so many requests for `Win32::ODBC` constant and function details that I am providing this information in Appendix B. Here you will find a wealth of information regarding what functions ODBC supports and what constants unlock the treasures that await an ODBC user.

## **Appendix C: Win32 Network Error Numbers and Their Descriptions**

With so many possible Win32 network errors, this appendix provides a programmer with a way to help figure out what a network-related error really means.

### **Note**

*Appendices B and C appear on the web site for this book at [www.roth.net/books/extensions/](http://www.roth.net/books/extensions/).*

## **Coding Style**

Almost all programmers have a programming style that they use. Perl programmers are no exception. Unfortunately, many of the styles used are not designed for clarity. This can be problematic for someone who is new to the language or who has to read through pages upon pages of source code.

I have heard many coders tell me that they don't prefer my style of coding for a variety of reasons, but then again, I have been told that my style is relatively easy to read. Because it is true that what one coder dislikes another coder adores, I think it is important for you to understand my style.

## Hungarian Notation

Generally speaking, I make use of the so-called Hungarian Notation programming style, in which the name of a variable begins with a prefix that indicates the type of data that the variable represents. For example:

```
$iTotal = 0;  
$szName = "my test text string";
```

Here, the `$iTotal` variable has a prefix of `i`, indicating that the variable represents an integer value. The other variable (`szName`) would represent a character string. In C parlance, the `sz` is indicative of a zero (or NULL) terminated string.

Now, all of the hard-core Perl readers are thinking, "Sure that is fine for C coding, but this is Perl!" Because Perl variables have the ability to morph themselves into whatever format they need to be in, prefixing the variable with a type indicator is not necessary. This is true. True, that is, until either a) your script becomes so large that it is considered a program, or b) other (possibly non-Perl) programmers have to read and modify your code.

It ends up that, for many of my clients, this style is absolutely invaluable. Some scripts that I have authored are large and are used in production environments by C++ coders. When they have to modify the Perl code, it is helpful to understand what the variable's intention was: a Boolean flag, a character string, a floating-point number, a reference (a.k.a. a pointer), and so on.

Another part of the Hungarian style is the way case plays a role in a variable. Basically, the beginning character of a word is capitalized, as in:

```
$iTotaLSoRtedEnTRies = 33;
```

If you were to not use mixed case, the variable would look like this:

```
$itotalsortedentries = 33;
```

I find it much easier to read the first example.

## Formatting

Throughout this book, you will run into different formatting styles. Each style has a meaning that is important to understand.

## Nonproportional Typeface

Generally speaking, filenames, variables, code segments, and the output of a script are listed in a nonproportional typeface such as:

- File: `c:\temp\MyFile.txt`
- Variable: `$iCountTotal`
- Code segment: `print "Welcome to Perl!\n";`
- Output: The file was unable to open.

References to functions or extensions are also in nonproportional typeface: You can find the `CreateUser()` function in the `Win32::NetAdmin` extension.

## Brackets

When listing a prototype for a function, optional parameters are listed in brackets (`[]`). This means that whatever is in the brackets is not required. Typically, you will see a comma in the brackets. This means that if the optional parameter is specified, the comma must also be specified:

```
ScheduleList( $Machine [ , \%List ] );
```

If a reference to a hash (`\%List`) is passed into the `ScheduleList()` function, then it must have the comma present, as in:

```
$Result = Win32::AdminMisc::ScheduleList( '\\\\server' , \%List );
```

If, however, the optional parameter is not passed in, then the comma is not specified:

```
$Result = Win32::AdminMisc::ScheduleList( '\\\\server' );
```

## Last Thoughts

Perl has been a fun language to learn, with all of its obfuscation and hidden tricks that keep you guessing. But it is also a fast language to code in. I was once in a room with two engineers who took more than 20 minutes trying to convince Microsoft Excel to properly format a histogram chart. I finally stopped what I was doing and wrote a simple script that not only formatted the data correctly but controlled Excel to manually create the chart. I was able to do this in less than 5 minutes. Both of the engineers' jaws dropped, and they both asked for copies of the script.

Perl is fast. Perl is quick. Perl is open source. Perl is not well documented. This book is here to help. I hope that this book serves you well and that you enjoy reading it as much as I have enjoyed writing it.

Cheers!

# Chapter 1. Why Perl on Your Win32 Machine?

All operating systems have ways of automating procedures and tasks. Usually this is done by collecting a series of commands in a file and having some way of executing these commands. Typically, this is performed by a program called a *shell*. UNIX users have been fortunate to have a myriad of shells to choose from: the C shell, the Bourne shell, and the Bourne Again shell (BASH), just to name a few. These shells are rich in function and can perform some complex command processing. You could even consider these shells programming environments. In the Microsoft operating systems, however, there has only been the command processor.

The *command processor* is the default shell for various versions of Microsoft DOS. This shell has had minimal functions for processing batched commands. This has forced administrators to write slightly complex programs in some languages such as C, Pascal, and BASIC. Only the most simple of tasks can be automated using the command processor's batching capabilities.

## History of Win32 Perl

Win32 platforms started to ship with a command processor that was modified to perform more complicated functions. These modifications, however, were far from sufficient for most administrators. It wasn't until Microsoft contracted with Dick Hardt's team at Hip Communications to port Perl to the Win32 platform that administrators had any hope of a decent freeware scripting utility. It was 1995 when the early versions of the Win32-compatible version of Perl started to find their way onto machines across the globe.

Not only did Hip Communications' Win32 Perl provide most of the functionality of the UNIX flavors of Perl, it also extended itself into the Win32 API. This provided an interface into the Win32-specific world of administration. Now it was possible not only to process Perl scripts, but also to access the computer's Registry, event logs, user databases, and various other features that are found only on Windows 95 and NT.

Since the Win32 Perl debut there have been many so-called builds, with each one fixing bugs and adding more functionality to its predecessor. Hip Communications continued to develop its version of Perl build by build. To be sure, there were other versions of Perl that would run on the Win32 platform. Some were ingenious enough to implement things that eluded Hip's port, like the `fork()` command (as the Win32 port from Mortice Kern Systems [MKS] provided).

The Hip port finally gained the acceptance of the group of programmers committed to developing different ports of Perl (the Perl Porters group). There were technical differences between the original version of Perl and the Win32 version (for example, Hip's port used C++ classes to encapsulate Perl's functionality). These differences kept the two versions of Perl (the original "core distribution" and Hip's Win32 Port) from merging together.

### Note

*More information regarding the Perl Porters group can be found at <http://www.perl.com/reference/query.cgi?porting>.*

Since Hip Communications released the original version of Win32 Perl, much has changed. Hip changed its name from Hip Communications, Inc., to ActiveWare. Then O'Reilly & Associates teamed with ActiveWare to create the ActiveState Tool Corporation. The Perl Porters group worked together with Hip (which later became ActiveWare and then even later yet renamed itself to ActiveState) to make the core distribution compile and run on the Win32 platform.

Beginning with Perl 5.005, ActiveState dubbed its Win32 build of Perl as *ActivePerl*. Basically, this was the core distribution version of Perl 5.005 but with the Win32 library of extensions included with the distribution package. Additionally, there are some built-in functions that are native to Win32 platforms only.

Starting with Perl v5.6 (aka Perl 5.006), both the ActiveState version of Perl and the core distribution have merged into one code base. This means that anyone can download the Perl v5.6 source code and compile it for Solaris, Macintosh, or Win32 (among other platforms). The Perl Porters group has done some incredible work to provide true cross-platform compatibility for version 5.6. This includes the much-sought-after features such as `fork()` emulation.

## ActiveState's ActivePerl

Most users will install ActivePerl and then start using it for all sorts of scripting needs. This means that these users will download the binary distribution of ActivePerl. Alternatively, many die-hard administrators and coders will download the Perl source code so that they can manually compile the source code themselves. Many organizations require this so that an inspection can occur to guarantee that there are no security or integrity problems. This also enables an organization to compile Perl with specific modifications that are not implemented with the binary distribution from ActiveState.

There are drawbacks, however, to manually compiling Perl. If you choose to compile Perl yourself, there could be complications. You need to keep in mind that any modifications you make (such as changing the standard C macro definitions) might result in binary incompatibility. This means that extensions that were compiled for the standard ActivePerl distribution might not work with your compiled version. You would have to recompile each extension you want to use.

Another problem with compiling your own copy of Perl is that there are a couple of extra components that ActivePerl distributes with its precompiled packages that are not available in source code form. These value-added components extend Perl beyond a simple scripting tool. These additions consist of an Information Server API (ISAPI) application called `PerlIS.DLL` and a Windows Shell Host engine known as *PerlScript*.

## Perl ISAPI Support

Many Web server administrators will configure their Web servers to run Perl-based CGI (common gateway interface) scripts. This effectively allows the Web server to run programs that interact with databases and dynamically display Web pages (among other things). Such CGI scripts, however, cause the Web server to create a new Perl process for each script executed. If 1,000 users accessed the script on the server at the same time, there would be 1,000 Perl processes running concurrently. You can imagine what type of performance impact that could incur. Ideally, there would be a way to increase performance but still provide full CGI-like functionality using Perl scripts. This is where the PerlIS ISAPI application comes in.

The [PerlIS.DLL](#) file is a clever ISAPI application that enables an ISAPI server (such as a Microsoft's IIS Web server) to quickly load and run Perl scripts. This application does not cause the script to run any more quickly than it normally would, but it does speed up the loading of the script because it can cache Perl scripts in memory. On busy Web servers, this extension can make a noticeable difference in CPU use.

PerlIS performs its magic by executing scripts in the same process as the Web server itself. Therefore, there is no new process created for each script that is run, resulting in less memory use and a smaller performance hit (because there is no overhead for creating new processes). Additionally, Perl scripts are cached in memory, so there is less time overhead to load the script from disk. What is even greater is that the script is cached in its compiled state; there is no need to recompile, so even that overhead is minimized.

Even though PerlIS is a wonderfully useful tool, there are some slight limitations. See [Chapter 12](#), "Common Mistakes and Troubleshooting," for more details.

## PerlScript Support

*PerlScript* is ActiveState's response to scripting languages such as VBScript and JavaScript. PerlScript is a Windows Shell Host (WSH) engine that can be used anywhere other WSH engines can be used. WSH scripts can run from Active Server Pages (ASP), from macro languages (such as MS Excel or MS Word macros), or even from the command line using the [WSCRIPT.EXE](#) WSH container program.

Because a WSH engine is really a COM server, any application that can interact directly with COM (for example, Visual Basic, C, Pascal, and so on) can create instances of PerlScript. For example, a C++ program could create an instance of a PerlScript object so that it can utilize Perl's regular expression capability and even interact with Perl extensions. Such functionality could be used to provide a program with awesome macro functionality (in the form of Perl scripts). An application can use a WSH engine for scripting like macros. Consider that Microsoft Word's macro language (Visual Basic for Applications) can instantiate a Perl script by utilizing the Perl WSH engine. This also means that an application could execute Perl scripts on remote machines by using DCOM.

PerlScript is implemented as the [PerlSE.dll](#) file. This is a COM server that must be registered properly before it can be used. If you are installing Perl by hand, you will have to register the COM object by running the [REGSVR32.EXE](#) application, as in:

```
regsvr32 c:\perl\bin\perlse.dll
```

See the section "[Installing Perl](#)" later in this chapter for more information.

## Perl Modules and Extensions

Perl is a very rich language just by itself. It has networking functions, regular expressions, and string, array, and hash manipulation functions. Perl can shell out and run other applications, open files, create pipes, and the list goes on. One of the most ingenious aspects of Perl, however, is the capability to extend the language. Perl allows itself to be extended in two ways: modules and extensions.

Collectively, these modules and extensions constitute Perl's libraries. Just as the C language has a series of libraries (stdlib, string, math, and so on), so does Perl. Such libraries are simply add-on functionality that is not built into Perl itself. Instead, they are separate pieces of code that can be added or removed from a Perl installation without affecting Perl itself. Although Perl will still function without these modules and extensions, scripts that depend on them will fail.

## Modules

*Modules* are just Perl code that you might want to use from time to time. These are also known as *packages*. If you appreciate and use certain functions (such as centering text on the screen or printing text backward), you could store these functions in a Perl module (a `.PM` file). Whenever you have a Perl script that needs these capabilities, your script would just issue the Perl `use` command, as in:

```
use MyModule;
```

or

```
use Win32::AdminMisc;
```

From that point on, the script could use any function stored in the module. It's quite simple, really. When creating a module (a `.PM` file), you need to follow two certain rules.

The first rule is that the module must begin with the following command:

```
package MyModule;
```

`MyModule` is replaced with the *case-sensitive* name of your module. This command can be followed by any Perl code that will initialize variables and perform any startup procedures you might want performed.

The second rule is that the end of the main block of code must end with the following command:

```
1;
```

This is a number 1 followed by a semicolon. It will result in the module returning a TRUE value (the number 1). This tells the `use MyModule` command that it successfully loaded the module. If, for whatever reason, the module does not initialize correctly, it could return a 0 value (which would cause the `use` command to fail, resulting in script failure).

## Extensions

Similar to modules are *extensions*. These are libraries of code written in another language (typically C). You can give Perl the capability to perform complex functions that it cannot perform by itself or that just would not be practical. If you wanted to have Perl compute pi to the thousandth decimal place, for example, it would be much faster to have a small C program do the work instead of Perl. Likewise, if you needed Perl to talk with a Sybase database server, you would want to write that

functionality in C and have Perl access those functions because Perl inherently does not know anything about Sybase servers.

Perl extensions consist of two separate files: a module (`.pm` file) and a `.dll` file. When Perl loads the extension (due to either a `use` or `require` command), Perl will load the `.pm` module first. The module will then *bootstrap* (load and initialize) the `.dll` extension. When this is done, the extension is considered to be loaded, and the Perl script can interact with it.

## Note

*Most Perl extensions consist of a Perl package (`.pm`) file and a related `.dll` file. Perl packages consist of `.pm` files, but an extension requires a `.pm` file in addition to a `.dll` library file.*

*Some extensions might require other files (such as additional `.dll`, `.ocx`, `.exe`, or others). If this is the case, these files must also be installed. Details about this usually can be found in the extension's `readme` file, documentation, or manual page.*

Extensions and modules are stored in Perl's library directories. Modules from the default namespace are located directly in the library directory. Modules that have their own namespace are located in subdirectories to the library directory.

For example, the `CGI.pm` module would be located in the library directory:

```
lib\cgi.pm
```

Another example would be the `File::Find` module, which is located in:

```
lib\file\find.pm
```

Extensions have a module file and an extension file. The module file follows the same rules as previously described. The extension file, however, is located in a different directory. Extension files are located in a path that describes the namespace and the extension name under a `lib\auto` directory.

For example, the `SDBM_File` extension has two files in the following directories:

```
lib\sdbm_file.pm  
lib\auto\sdbm_file\sdbm_file.dll
```

Another example is the `Data::Dumper` extension. Its files are located in:

```
lib\data\dumper.pm  
lib\auto\data\dumper\dumper.dll
```

## Namespaces

Each module and extension defines a *namespace*. This is a place where information regarding the module or extension is stored. A script can access a particular namespace by referencing it directly. For example, if you wanted to access a `$ModulesLastError` variable from the `MyModule` namespace, you would access it by specifying the namespace followed by double colons (::) and then the variable name, as in:

```
$ErrorValue = $MyModule::ModulesLastError;
```

This instructs Perl to look into the `MyModule` namespace and retrieve the value from its `$ModulesLastError` variable.

When you run a script, by default, it stores its variables in the `main` namespace. Therefore, variables created in a simple script should always specify the `main` namespace, as in [Example 1.1](#).

### **Example 1.1 Referring to the `main` namespace**

```
01. my $main::MyVar = 12;
02. print $main::MyVar, "\n";
```

A shortcut to referring to the `main` namespace is to specify only the double colons. [Example 1.2](#) illustrates this.

### **Example 1.2 Referring to the `main` namespace using a shortcut**

```
01. my $::MyVar = 12;
02. print $::MyVar, "\n";
```

Many programmers are not aware of namespaces and are surprised to learn of their capabilities. This is because of the concept of the *current namespace*. A piece of code can reside in any namespace (Perl's `main` namespace or some module/extension's namespace). If the code refers to a variable or function that also resides in the same namespace, there is no need to prepend the name of the namespace to variables and functions. It is assumed that you are talking about the current namespace.

For example, consider [Example 1.3](#). Line 2 sets the `$MyVar` variable to the value of 12. Because there is no namespace specified, the variable is assumed to be in the *current namespace*. Therefore, the value of 12 is printed in both lines 8 and 10 because they both refer to the `$MyVar` variable in the `main` namespace (line 8 is actually referring to the current namespace's `$MyVar` variable).

Notice, however, that lines 3 through 5 actually call the `GetAnyDomainController()` function in the `Win32::NetAdmin` extension. Because each extension and module defines its own namespace, it, too, has a name-space where it can store variables. And that is exactly what these lines do. Notice that, in line 5, the variable that will be set with the name of a domain controller machine is from the extension's namespace (`$Win32::NetAdmin::MyVar`). This is why line 12 prints a different value than lines 8 and 10.

### **Example 1.3 Referring to different namespaces**

```

01. use Win32::NetAdmin;
02. $MyVar = 12;
03. if( Win32::NetAdmin::GetAnyDomainController( '',
04.                               '',
05.
$Win32::NetAdmin::MyVar ) )
06. {
07.     print "The current namespace's ('main') MyVar variable is: ";
08.     print $MyVar, "\n";
09.     print "The main namespace's MyVar variable is: ";
10.    print $main::MyVar, "\n";
11.    print "The Win32::NetAdmin namespace's MyVar variable is: ";
12.    print $Win32::NetAdmin::MyVar, "\n";
13. }

```

## Win32 Perl Extensions

Many independent coders have written extensions that only run on Win32 platforms. Such extensions range from an interface into dynamic link library files (DLLs) to remote access service (RAS) administration functions. These extensions provide an absolute wealth of functionality for anyone who administrates or even just uses Microsoft's Win32 platforms.

### Note

A DLL can have any mix of functions and resources. A resource can be an icon, a wave sound, a font, a dialog box, a menu bar, or a graphic, just to name a few.

Some of the most valuable extensions are found on the Comprehensive Perl Archive Network (CPAN) under the Win32 modules section (<http://www.perl.com/CPAN-local/modules/by-module/Win32/>). CPAN is a wonderful place to grab modules. Quite often, however, authors have updated versions of a module or even new extensions (alpha, beta, and release versions) available on their Web sites. [Table 1.1](#) documents some worthwhile Web sites for information on Win32 Perl.

**Table 1.1. Online Resources for Win32 Perl**

Site	URL	Contents
ActiveState's Web site	<a href="http://www.activestate.com">http://www.activestate.com</a>	The ActiveState version of Win32Perl and its standard extensions. These extensions come with ActivePerl.
Perl.com	<a href="http://www.perl.com">http://www.perl.com</a>	The core distribution version of Perl and other modules.
Aldo "dada" Calpini's Perl Lab	<a href="http://dada.perl.it">http://dada.perl.it</a>	Win32::API Win32::Internet Win32::Clipboard Win32::Console Win32::GUI Win32::Sound Win32::Shortcut

**Table 1.1. Online Resources for Win32 Perl**

Site	URL	Contents
Roth Consulting's Perl Pages	<a href="http://www.roth.net/perl/">http://www.roth.net/perl/</a>	Win32::AdminMisc Win32::Daemon Win32::EventLog::Message Win32::Message Win32::ODBC Win32::Perms Win32::Pipe Win32::RasAdmin Win32::Tie::Ini
Jutta Klebe's Web site	<a href="http://www.bbyte.de/jmk/">http://www.bbyte.de/jmk/</a>	Win32::PerfLib
GoneFishing.org	<a href="http://www.gonefishing.org/techstuff/">http://www.gonefishing.org/techstuff/</a>	Win32::FileSecurity
CPAN	<a href="http://www.cpan.org/modules/by-category/22_Microsoft_Windows_Modules/Win32/JHELBERG/">http://www.cpan.org/modules/by-category/22_Microsoft_Windows_Modules/Win32/JHELBERG/</a>	Win32::Lanman

## Win32 Extensions Are DLLs

Creating an extension is usually a matter of writing a collection of functions and storing them in a library of some sort. In the Win32 world, these extensions are in the form of a DLL. Win32 users are most likely familiar with DLLs because all Windows machines are filled with them. Files like `KER-NEL32.DLL`, `MSCRT.DLL`, or `VBRUN300.DLL` are riddled throughout the hard drive. These files are, in the simplest of terms, a library of functions and resources. Such functions and resources are available to any program that chooses to use them.

Imagine you write a Web browser program. This program must be able to open connections to other computers across a TCP/IP network such as the Internet. The code to perform these network activities can take up quite a bit of hard disk space. Assume, for example, that this program is 500KB in size. Of this size, also assume that 50KB is the network code. Now let's say you write an email program. This program must also use the same 50KB of network code. Then you write an FTP program that accesses computers over the Internet, so it will, like the other two, need the same 50KB piece of network code.

Between these three programs, 150KB of your hard drive is taken up with the same code. This is a big waste of valuable drive space. It would be a great if you made a library of only these networking functions. All your programs could just load the library and call the functions from that library. This way, you only take up 50KB of your hard drive with the needed network code. This is the idea behind a DLL. Instead of having all network functions statically linked into each of your programs, they dynamically link to your single library of network code. This is exactly what the `WINSOCK.DLL` and `WININET.DLL` files are.

One of the biggest advantages of using DLLs is that they are discrete libraries of code. This means that a DLL is independent from other libraries or programs. Suppose some program has a bug in it, and the author has to fix it. If the bug is in the DLL, the author needs only to send the fixed version of the DLL to his clients. There is no need to send out the entire program again because only the DLL was affected. Likewise, if a program has a bug in it, only the program needs to be fixed and

replaced, not necessarily any of its DLLs. Quite often, you will see updates to common DLLs such as [VBRUN300.DLL](#) (the Visual Basic Runtime library version 3.00—a DLL that is required for programs written using Visual Basic).

Win32 Perl extensions are just DLL files. They contain code that can be called from a script. You might notice file names such as [Netresource.DLL](#) and [OLE.DLL](#) littered throughout your Perl directories.

## Note

*For users of the older ActiveState version of Perl (before version 5.003 build 400), extension files have a .pll instead of a .dll filename. For example, they might be [NeTResource.pll](#) and [OLE.pll](#), presumably for Perl linked library.*

*Although these older extensions end with .pll, don't be fooled; they are just DLLs that have been renamed. However, even though these are simply renamed .dll files, they cannot be interchanged with extensions built for older versions of Perl. That is, a .pll extension cannot be renamed and used with ActiveState's v5.3 build 400+, core distribution (version 5.004), or ActivePerl (version 5.6). If you tried this, your Perl script would fail with an error message.*

# Different Perl Libraries

One of the first things you need to be familiar with is the Perl library directory. Perl maintains a directory of library files that are important and necessary for Perl. You might never write a script that uses any of these libraries, but you might download a script that will. Unlike its UNIX-like relatives, Win32 Perl's directory names are not case sensitive. In other words, a directory that is named Perl could be spelled PERL, PerL, perl, or pErL. This is a feature of Win32, not Perl itself.

Older versions of Perl used only one library in which all modules and extensions resided. More recent versions of Perl make use of two different libraries: the *default* and *site* libraries. The default library is used for those modules and extensions that are platform agnostic. The site library is used for extensions that are specific for your particular site and platform. Within each library, there are two different paths you need to understand.

## Default Library

The default library directory is located in the `perl\lib` directory. If you have installed Perl in `c:\perl`, then the default library directory would be `c:\perl\lib`.

All of the default libraries that are general enough to apply to any installation of Perl are stored in this library path. Many common modules, such as the Net modules (for example, [Net::DNS](#) and [Net::Ping](#)), LWP modules, Getopt modules, and Time modules, are located in the default library.

## Site Library

The site library contains modules specific to your particular site. This includes modules that are platform specific (such as all of the Win32 extensions) and modules that only apply to your specific installation.

The site library is located in the `perl\site\lib` directory. If you have installed Perl in `c:\perl`, then the site library directory would be `c:\perl\site\lib`.

## Installing Extensions

After you have Perl and its Win32 extensions installed, you are ready to create Perl scripts that will make your life with Windows a bit easier.

If you have to install an extension, however, things can be a bit confusing. Sometimes when you download an extension, you might end up with many files and very vague documentation on how to install the extension.

There is really nothing to be concerned about, however. The process is very easy and makes total sense after you have walked through it.

Unless you have downloaded source code for an extension, the process of installing is usually just a matter of copying files to proper locations on your hard drive. If you have downloaded source code, you should probably read [Chapter 10](#), "Writing Your Own Extension." If you have downloaded a binary (already compiled and built) version of the extension, all you need to know is where the files go.

## Directory Paths

The `AUTO` directory is a special directory found in both the default and site libraries. This is where binary extension (`.dll`) files are located. This directory is a subdirectory of the root in each library (for example, `SITE\LIB\AUTO`).

The way that the directory layout works is that a module's `.PM` file is stored in a directory that represents its namespace. For example, the `Win32::EventLog::Message` extension would store its module file (`Message.pm`) in the following path:

`PERL\BSITE\LIB\WIN32\EVENTLOG\Message.pm`

The actual binary extension file (`Message.dll`) would be located in a similar path under the `AUTO` subdirectory; however, the path would also include the module's name:

`PERL\BSITE\LIB\AUTO\WIN32\EVENTLOG\MESSAGE\Message.dll`

Typically, when Perl loads a module or extension, it first checks the default library. If that fails to find the module or binary extension, it tries again using the site library.

### Note

*It is possible to accidentally have installed the same extension or module multiple times. It is fairly important to check for such multiple installations of a particular module or extension. This is regardless of how you install it.*

*Consider a situation in which you have already installed some module in the default library (`perl\lib`), but then you install an updated version of the module into the site library (`perl\site\lib`). In this case, Perl would not use the updated version because it normally first checks the default library before looking into the site library.*

*If you have recently installed an updated version of an extension (or module) but your scripts do not see the updated features of the extension, check for multiple copies of the extension across the different libraries.*

There are basically two ways to install a Perl extension:

- Manually
- Using the Perl Package Manager (PPM)

### ***Manual Installation***

Traditionally, a user would have to download an extension and manually install it. Usually the extension would be stored in a `.zip` or `.tar` file. You would need to first unarchive the files and then move them into their appropriate directories. This might mean you have to create some directories.

For example, let's say you download the new `Win32::EventLog::Message` extension. Because the extension is Win32 specific, it belongs in the site library. Therefore, you would unzip the contents of the archive and then do the following:

1. Create a `perl\site\lib\win32\eventlog` directory.
2. Copy the `Message.pm` file into the new `perl\site\lib\win32\eventlog` directory.
3. Create a `perl\site\lib\auto\win32\eventlog\message` directory.
4. Copy the `Message.dll` file into the new `perl\site\lib\auto\win32\eventlog\message` directory.

The extension would then be installed. Of course, read any of the readme files to make sure there are no other files that need to be copied anywhere.

### ***Perl Package Manager (PPM)***

The *Perl Package Manager (PPM)* is an easy way to automatically download and install extensions and modules. PPM comes with ActivePerl and works by accessing Perl Package Descriptor (`.ppd`) files. These files are XML documents that describe a package or extension. They describe the author, function, version, and path (among other things) of a particular package.

PPM is designed to connect over the net to a package repository and download an appropriate `.ppd` file. After the `.ppd` file has been obtained, the Package Manager will automatically install the package by copying `.pm`, `.dll`, help, and any other files to their appropriate locations. PPM will then update configuration-tracking files so that, when you run the manager in the future, it knows what it has done. This makes it rather quite easy to install new or update previously installed Perl packages.

By default, PPM is configured to point at ActiveState's Perl package repository. It maintains a list of commonly used Perl packages including Win32 specific packages.

## Note

*There have been several bugs found in the Perl Package Manager. Quite often, an attempt to use PPM will result in either a failure to install the extension or the script will prematurely break and display some error.*

*If you experience such problems, check out ActiveState's Web site (<http://www.activestate.com/>) for any bug fixes or updates to PPM.*

PPM enables you to install, remove, query, verify, and search for packages. All of these commands can be issued from the command line by passing the command in followed by any parameters required for that particular command.

Because PPM can install extensions from the World Wide Web, it makes installing and updating extensions quite easy.

## Tip

*There are two different ways to run PPM: interactively or from a command line. Running PPM interactively provides a rich environment in which to manage various Perl packages (modules and extensions). For more information on running interactively, either consult the PPM help files or run the manager (run PPM without specifying any parameters) and type in 'help'.*

## **Installing from the Web**

The PPM script can be used to automatically download and install a given extension or module. This is done by running PPM from a command line passing in the install keyword followed by the name of the extension.

## Note

*Take note that, due to syntax limitations, all double colon (:) characters are replaced with a single dash (-) character.*

*For example, if you wanted to install the Win32::NetAdmin extension, you could run:*

```
ppm install win32-netadmin
```

There are several Perl package *repositories* available on the Web. These are Web or FTP sites where PPM-based .ppd files and their archives reside. By default, PPM is configured to use ActiveState's Perl package repository. You can modify PPM's configuration to include other

repositories as well by running PPM interactively and using the `set repository` command. Refer to PPM's documentation for more details.

You can also temporarily specify a repository by passing it into PPM using the `--location` keyword. For example, to install the `Win32::Daemon` extension from Roth Consulting, try the following:

```
ppm install win32-daemon --  
location=http://www.roth.net/perl/packages/
```

Alternatively, you can specify a path (or URL) to a particular `.ppd` file. The preceding example could be rewritten as follows:

```
ppm install http://www.roth.net/perl/packages/win32-daemon.ppd
```

Either of these would work just fine. However, the latter suggests that you can specify a `.ppd` file locally. Let's say a friend sent you the `.ppd` file through email. Let's also say you saved the file to your `c:\temp` directory. To install the extension, you could use the following command:

```
ppm install c:\temp\win32-daemon.ppd
```

Running this command might result in files being downloaded using HTTP or FTP. This depends on how the `.ppd` file is constructed.

## **Using Proxy Servers**

If you make use of HTTP proxy servers to get out onto the Internet, you will have to inform PPM of this. Typically, users behind firewalls (such as corporate users) will have to contend with proxy servers. It is easy enough to configure PPM to use a proxy server.

The most flexible way to do this is by modifying an `HTTP_Proxy` environment variable. This can be either a user or system environment variable. It also can be set computer wide, or it can be set using the DOS `set` command, which would affect only the DOS window; when the DOS window is closed, the environment variable is lost.

To illustrate how you can set a proxy server, simply open a DOS window (on Windows NT/2000 run `cmd.exe`, and on Windows 95/98/ME run `command.com`) and then enter the following:

```
set HTTP_Proxy=http://myproxy.mydomain.com:8080
```

This will force PPM to always connect to the proxy server <http://myproxy.mydomain.com> on port 8080 when it is downloading Perl packages.

## **From Local `.ppd` File**

If you are having trouble downloading a Perl package, you can bypass the network altogether. There are two ways to do this: One requires only the `.ppd` file, and the other requires both the `.ppd` file and the package's archive file.

If you have the package's `.ppd` file (for example, someone emailed it to you or you downloaded it earlier) you can manually install it by passing in the path to the `.ppd` file:

```
ppm install c:\temp\win32-netadmin.ppd
```

This will result in PPM reading the specified `.ppd` file in the `c:\temp` directory. The contents of this file will cause the `Win32::NetAdmin` archive package to be downloaded from a package repository (most likely from the Internet) and automatically installed. If you already have the archive file (typically a `.zip`, `.tar`, or `.gz` file), you can edit the location of the archive in the `.ppd` file. You can modify the http URL to point to the local path (such as `c:\temp\win32-netadmin.tar.gz`).

## Installing Perl

The ActivePerl zip-file-based distribution can be used to manually install Perl, but it requires some configuring on the administrator's part. Configurations include modifying the `%PATH%` environment variable, associating the `.pl` file extension with the Perl executable and registering the various COM-based Perl components (such as the PerlScript Windows Shell Host engine).

ActivePerl's Microsoft Windows Installer (MSI) package is designed to work with Windows machines that have the MSI installer services (such as Windows ME, Windows 2000, and older versions with MSI services installed). This automates installing Perl, registering system components, modifying environment variables, and any other modifications that need to be made. Although this option is very easy to use, it comes at the cost of flexibility. An administrator has less control over how Perl is installed when using the MSI package.

Additional information regarding modules and extensions, not to mention Win32 Perl itself, can be found at various Web sites. Many well-known users have devoted time and resources to providing information and maintaining their Perl-related Web sites:

- **Joe Casadonte's Web site:** <http://www.northbound-train.com/perlwin32.html>
- **Tye McQueen's Web site:** <http://www.metronet.com/~tye/>
- **Philippe Le Berre's Web site:** <http://www.le-berre.com/>
- **LibWin32 Documentation:** <http://theory.uwinnipeg.ca/CPAN/by-name/libwin32.html>
- **Matt Sergeant's Database FAQ:** <http://www.geocities.com/SiliconValley/Way/6278/perl-win32-database.html>
- **ActiveState's Win32 Perl FAQ:** <http://www.activestate.com/support/faqs/win32/>
- **Robin's Perl for Win32 Page:** <http://www.geocities.com/SiliconValley/Park/8312/>
- **Carvdawg's Perl Page:** <http://patriot.net/~carvdawg/perl.html>

There has been, and will continue to be, a huge influx of Win32 extension development. What is the justification for developing Win32 extensions? The following section defines Win32 extensions and explains why more people are taking advantage of the Perl environment for Win32 tasks.

## Loading Win32 Extensions

When you are going to make use of a Win32 extension (or any module or extension, for that matter), you need to first use it in your module. The `use` command will load the extension into memory:

```
use Win32::Pipe;
```

Consult a Perl language reference for a full explanation of the `use` command, such as the de facto standard *Programming Perl* by Larry Wall, Tom Christiansen, and Randal Schwartz (O'Reilly & Associates).

After the `use` command loads the extension, your script is ready to call functions within the extension.

### Tip

*Many of the default Win32 extension's functions do not require that a script explicitly be loaded a `require` or `use` command. Some of the module's functions are automatically loaded when Perl is launched. You can call these particular Win32 functions, such as `Win32::LoginName`, without ever having to load the Win32 module. Refer to the section "[Miscellaneous Win32 Functions](#)" in [Chapter 9](#) for more details.*

## Methods Versus Functions

Some extensions are just collections of functions that perform a task. The `Win32::NetAdmin` extension is a perfect example. You can call this extension's functions, which will perform some task and return a result of some sort. A function call into an extension requires that the full namespace be provided, as in [Example 1.4](#).

### Example 1.4 A function call

```
01. use Win32::NetAdmin;
02. my $UserName = "Joel";
03. if( Win32::NetAdmin::UsersExist( '',    $UserName    ) )
04. {
05.     print "$UserName is a user.\n";
06. }
07. else
08. {
09.     print "$UserName is not a user.\n";
10. }
```

Other extensions are collections of functions that must be used together. Along with these functions are variables that must be kept in association with the functions.

An example of this is the `Win32::ODBC` extension. When you use `Win32::ODBC`, you are generally connecting to a database. If you need to connect to two separate databases, you must keep two separate collections of variables. This can be accomplished using *object-oriented programming*

(OOP). More information regarding Perl OOP can be found in Perl's manual pages and in Damian Conway's *Object Oriented Perl* (Manning Publications).

The `Win32::ODBC` extension enables you to create an object using the `new` command. This object is a collection of variables and functions that all work together. The new object acts like a reference to a hash, which contains functions known as *methods*. These methods can modify variables that reside only within the object. Such variables are called *members*. An object's methods and members all work together as if they all existed in the same namespace. This means that if a member is created in an object, any method can alter it. Unlike a module or extension's function, which requires calling the full namespace such as `Win32::ODBC::GetDSN()`, methods and members are referenced using the arrow operator (`->`) as if you were dereferencing a key from a hash reference, as in [Example 1.5](#).

### **Example 1.5 An object's members and methods**

```
01. use Win32::ODBC;
02. my $db1 = new Win32::ODBC( "Financial Database" ) || die;
03. my $db2 = new Win32::ODBC( "Order Database" ) || die;
04. # print the $db1 object's 'DSN' member (variable)
05. print $db1->{ 'DSN' }, "\n";
06. # Call the $db2 object's 'Connection' method.
07. print $db2->Connection(), "\n";
08. print $db2->{ 'DSN' }, "\n";
09. print $db1->Connection(), "\n";
```

### **Note**

*Throughout this book, references are made to variables, functions, members, and methods. It is important to note the difference because calling directly into a method as if it were a function could result in incorrect results, if not even a script-stopping error.*

Some extensions' functions can be called as both functions and methods. Their results usually depend on the context in which they were called.

Most extensions are documented either in a readme file that comes with the extension, as embedded POD code in the module file, or on a Web page somewhere on the Internet. Such documentation traditionally describes the module or extension's variables, functions, members, and methods.

### **Plain Old Documentation (POD)**

*Plain Old Documentation (POD) is HTML-like code that many authors embed within a module. This convenience provides users with the documentation as long as they have the module file (like `NetAdmin.pm`). You can download certain scripts that parse out the POD code and convert it to text, HTML, RTF, and various other formats. The script `POD2HTML.PL` is one such script that comes with the core distribution of Perl.*

# Parse Exceptions

ActiveState releases new builds of Win32 Perl every so often. Each new release corrects problems from previous builds and implements new or updated functionality. Occasionally, a new build will be released that has changed the way Perl works internally. These changes are invisible in scripts but will break compatibility with Perl extensions.

ActiveState Perl users who have upgraded from one build to another might run into this problem. You'll know that this is the case because you will see the dreaded Parse Exception error. This error has been the subject of countless emails and Usenet postings.

Simply stated, a Parse Exception error takes place when you are trying to use an extension that was created for a version of Perl you are not using. If you were using ActiveState's Perl build 522 and then upgraded to build 620, for example, you might have seen this error message:

Error: Parse Exception.

What you would need to do is find a version of the extension that has been compiled with the header files and libraries from build 620. Usually the maintainer of an extension will make several builds available for just this reason—one for the core distribution and a few for different ActiveState builds.

The good news is that, when you upgrade to another build of the ActiveState build, all the standard extensions (including the Win32 extensions) are updated as well. It is all part of the downloaded archive. You need only to update any nonstandard extensions—that is, any extension that does not come with Win32 Perl (for example, [Win32::GUI](#)).

## Warning

*Make sure that, when you unarchive Perl, filenames are not altered. Some older versions of PKUNZIP and other dearchiving programs do not know how to handle long file names. This can result in the mangling of some very important files.*

*When running your scripts, if you are running into an error that indicates you are unable to find DYNALOADER.PM or some other file with a filename greater than eight characters, it might be that, when you unarchived Perl, the file DYNALOADER.PM was renamed to DYNALO~1.PM by a program that did not know how to save a long filename.*

*If you find yourself in this situation, either reinstall Perl using a more modern version of PKUNZIP (such as WINZIP or the like) or manually rename all the files. Because you might not know what the real filenames are supposed to be, it is probably best to reinstall with a newer unzip program.*

*More recent versions of Win32 Perl come as self-extracting and auto-installing executable programs. This takes care of all the details for you.*

## Error Handling

Generally speaking, when Perl encounters an error, it sets the error variables (`$_!`) to both the error number and the textual error message. This enables the script to determine the nature of a Perl-

generated error. The `$!` variable usually reports system errors. The `$^E` variable reports Win32 errors. However, most modules and extensions generally do not touch this variable.

Instead, they use their own way of error reporting, which makes it difficult for a coder to discover error information because the technique differs from extension to extension.

The Win32 extension defines two very important error functions: the `GetLastError()` and `FormatMessage()`.

## **GetLastError() Function**

The `Win32::GetLastError()` function returns an error code for the last error that the Win32 API generates. It is important to keep in mind that the error number returned by this function is a Win32 API error, not a Perl error. The Win32 API will generate some errors of which Perl will not be aware.

Win32 API errors are generated when a task fails that requires the operating system to become involved. When Perl opens a file, for example, it passes the request to the operating system. If the OS cannot open the file, it will generate its own error, which can be retrieved with `Win32::GetLastError()`. The OS then tells Perl that the file could not be opened, and Perl registers an error. [Example 1.6](#) demonstrates this.

### ***Example 1.6 Demonstration of the difference between a Perl error and a Win32 API error***

```
01. use Win32;
02. # Reset any Perl error
03. $! = 0;
04. # Purposely generate an error...
05. if( ! getpeername( "ThisIsABadPeerName" ) )
06. {
07.     # Get the error number
08.     my $Error = int( $! );
09.     my $Win32_Error = Win32::GetLastError();
10.    print "Perl error number: $Error\n";
11.    print "Perl error message: $!\n";
12.    print "Win32 error number: $Win32_Error\n";
13.    print "Win32 error message: " .
Win32::FormatMessage( $Win32_Error );
14. }
```

[Example 1.6](#) uses the `getpeername()` function because it best illustrates how Perl sees an error but Win32 does not. The two error numbers and error text might differ because they come from different sources.

The following output from [Example 1.6](#) shows how Perl might recognize that an error has taken place even though the Win32 API is unaware of any such error:

```
Perl error number: 9
Perl error message: Bad file descriptor
Win32 error number: 0
```

```
Win32 error message: The operation completed successfully.
```

Generally, this means that you should rely on the `$!` error variable only when the error was generated using standard Perl functions. If an error is generated from an extension, you can generally use the `Win32::GetLastError()` function to discover the nature of the error.

Another error variable is available that is designed to reflect Win32's errors. This is the `$^E` variable. You can refer to this variable much as you do `$!`, except that the error numbers it reflects is based on Win32's interpretation of an error as opposed to Perl's interpretation.

Both the `$!` and `$^E` error variables are overloaded such that when they are referred to in a numeric context, they represent an error number value. However, when referred to in a text context, they represent a text string describing the error. [Example 1.7](#) illustrates how you can reference these different variables in different contexts.

### **Example 1.7 Referring to the `$!` and `$^E` error variables**

```
01. # Reset any errors
02. $! = $^E = 0;
03. if( open( FILE, "< SomeFileThatDoesNotExist.txt" ) )
04. {
05.     print "The file successfully opened.\n";
06.     close( FILE );
07. }
08. else
09. {
10.     print "Could not open the file.\n";
11.     print "The Perl error number is: ", int( $! ), "\n";
12.     print "The Perl error text is: $!\n";
13.
14.     print "The Win32 error number is: ", int( $^E ), "\n";
15.     print "The Win32 error text is: $^E\n";
16. }
```

Both of these variables can be set to any arbitrary value. This means that if your script determines that an error occurred and obtains the error number (such as calling `Win32::GetLastError()`), it can set the error variable, such as:

```
$! = $^E = Win32::GetLastError();
```

The `Win32::GetLastError()` function takes no parameters and returns a numeric value that represents the previous Win32 API error message that the script generated. You cannot use this function to retrieve Win32 API errors generated by other running scripts, only the currently running one.

### **FormatMessage() Function**

After you have retrieved the Win32 API error number (see [Example 1.6](#)), you can request that the OS give you a human-readable version of the error with the `Win32::FormatMessage()` function:

```
$ErrorText = Win32::FormatMessage( $ErrorNum );
```

The only parameter passed in (`$ErrorNum`) is a numeric value (usually obtained with the `Win32::GetLastError()` function) that represents a Win32 API error number. The function will return a string that is the text-based description of the error message.

It is common to see the function `Win32::GetLastError()` passed in to `Win32::FormatMessage()`, as in [Example 1.8](#). This is a convenient and quick way to obtain the Win32 error text.

### ***Example 1.8 Subroutine for printing Win32 errors***

```
01. use Win32;
02. sub PrintError
03. {
04.   print Win32::FormatMessage( Win32::GetLastError() );
05. }
```

A much easier way to do this is to assign the error number to the `$^E` variable. Whenever the variable is referred to in a nonnumeric context, Perl will automatically perform the `FormatMessage()` conversion. This means that [Example 1.8](#) can quite easily be rewritten to be [Example 1.9](#).

### ***Example 1.9 An updated subroutine for printing Win32 errors***

```
01. sub PrintError
02. {
03.   $^E = Win32::GetLastError();
04.   print $^E, "\n";
05. }
```

## **Specific Win32 Extension Error Handling**

Unfortunately, you cannot use the `Win32::GetLastError()` function with all the Win32 functions. Some extensions require you to use specific error functions for a variety of reasons. This requirement means that you can use the `Win32::GetLastError()` function when you use these extensions. If your call to a particular extension's function fails, however, you should not use `Win32::GetLastError()` to discover why it failed. Instead, you should use whatever error function the extension provides. You will have to check the documentation for that particular extension to determine how to handle such errors.

To help clarify this issue, it is necessary to understand why this error madness exists. When a Win32 API function (such as retrieving a list of user accounts) fails, two things happen. First, the function itself returns an error indicating the nature of the failure. It is up to the extension to manage such knowledge. Second, the Win32 API internally stores the error. When you call `Win32::GetLastError()`, it is this stored error number that is returned. Suppose some Win32 extension's function discovers that an error occurred, and it handles it by making other Win32 API function calls in an attempt to correct the problem. Each time these additional API calls are executed, Win32 may internally store their results. By the time the extension's function returns to your code and you call `Win32::GetLastError()` to determine why the error occurred, you might

be retrieving the error information for a totally different function. This is why most extensions provide their own error-retrieval function.

Some Win32 extensions (such as `Win32::EventLog`) attempt to set the Perl error variable `$!` when it generates an error. It is *not* a good idea to depend on using `$!` under these circumstances.

Although the error variable is set to the correct Win32 API error number, the error text message will be incorrect.

Suppose a Win32 extension tries to load a file but fails. Then the extension sets the `$!` variable with the Win32 error number that was generated, which is 3 (the system could not find the path specified). If your script then prints the error text using the `$!` variable, as in the following:

```
print "Error: $!\n";
```

Perl would print the error text for error number 3, which is `No such process`. Perl error numbers are not the same as Win32 error numbers. So if a Win32 extension sets `$!` based on the results of a Win32 function or with the value of `Win32::GetLastError()`, you should avoid relying on `$!` to correctly report the error. This is another reason why `Win32::GetLastError()` is so important when using the Win32 extensions.

With so many Win32 extensions, many different ways of interpreting and managing errors exist. Four extensions, however, beg for a bit more attention than their documentation provides:

`Win32::NetResource`, `Win32::ODBC`, `Win32::RasAdmin`, and `Win32::WinError`.

### **Win32::WinError Error Handling**

The `Win32::WinError` extension exports several constants that can be used when determining the nature of an error. There is some question as to how useful this extension is because it maps error constants with error numbers but not the other way around. The following code will print the error code for the constant `ERROR_LOGIN_WKSTA_RESTRICTION`, for example:

```
use Win32::WinError;
print "The error number " . ERROR_LOGIN_WKSTA_RESTRICTION . "
represents the
error ERROR_LOGIN_WKSTA_RESTRICTION.\n";
```

The real benefit with this extension is in using it for the sake of testing the result of an error, as in [Example 1.10](#).

#### **Example 1.10 Using Win32::WinError**

```
01. use Win32;
02. use Win32::WinError;
03. # ...process some code...
04. # Test for an error...
05. my $Error = Win32::GetLastError();
06. if( ERROR_LOGIN_WKSTA_RESTRICTION == $Error )
07. {
08.     #...process the error...
09. }
```

```

10. elsif( ERROR_ACCOUNT_LOCKED_OUT == $Error )
11. {
12.     #...process the error...
13. }

```

This is useful when using the `Win32::API` extension.

## **Win32::ODBC Error Handling**

When a `Win32::ODBC` function fails, the only way to discover the error number is by using the `Error()` function. This is implemented as both a function and a method:

```

Win32::ODBC::Error();
$db->Error();

```

This function/method takes no parameters and will return either an array or a text string, depending on what the caller's context is.

Notice that this particular call can be either a function (`WIN32::ODBC::Error()`) or an object's method (`$db->Error()`). [Example 1.11](#) shows both uses.

### **Example 1.11 Using Win32::ODBC's Error() function and method**

```

01. use Win32::ODBC;
02. my $db;
03. if( !( $db = new Win32::ODBC( "MyDSN" ) ) )
04. {
05.     print "Unable to connect to the database.\n";
06.     print "Error: ". Win32::ODBC::Error();
07. }
08. elsif( $db->Sql( "SELECT * FROM Foo" ) )
09. {
10.     print "The query produced an error.\n";
11.     print "Error: ". $db->Error();
12. }

```

In an array context, this will return the elements in [Table 1.2](#). Each of the array's elements represents discrete information regarding the error.

**Table 1.2. Array Returned by Win32::ODBC::Error()**

<b>Index</b>	<b>Name</b>	<b>Description</b>
0	<code>ErrorNum</code>	Error number
1	<code>ErrorText</code>	Text description of the error
2	<code>Conn</code>	Connection number (this is, a value that is internally used by the extension)
3	<code>SQLState</code>	The SQLState of the error

In [Example 1.12](#), an error array is returned in which the third element (the SQLState of the error) is compared to a particular value. By returning an array, the script can process each aspect of an error.

### ***Example 1.12 Extracting information from the error array***

```
01. use Win32::ODBC;
02. my $db = new Win32::ODBC( "MyDSN" )
03.           || die "Can not connect: " ..
Win32::ODBC::Error();
04. if( $db->Sql( "SELECT * FROM Foo" ) )
05. {
06.     my @Error = $db->Error();
07.     if( "HYT00" eq $$Error[3] )
08.     {
09.         print "The database server took to long to process the
query.\n";
10.    }
11.    else
12.    {
13.        print "The query produced an error number $Error[0].\n";
14.        print "The error text is $Error[1].\n";
15.        print "The connection number is $Error[2].\n";
16.        print "The SQLState of the error is $Error[3].\n";
17.    }
18. }
```

### ***Win32::NetResource Error Handling***

There are two different functions to retrieve error information from the `Win32::NetResource` extension: `GetError()` and `WNetGetLastError()`. Don't be fooled, however; these functions are not interchangeable. When a `Win32::NetResource` function fails, you can retrieve the error with the following:

```
Win32::NetResource::GetError( $Error );
```

This function always returns a `1`, and the error number will be stored into the scalar that is passed into it (in the example here, the `$Error` variable). If the scalar `$Error` contains the value of `1208` (this is, the same value as the constant `ERROR_EXTENDED_ERROR` from the `Win32::WinError` extension), you need to get the extended error information. This happens when a particular network provider (for example, Microsoft LAN Manager or Novell NetWare) other than Win32 has reported an error. Some systems might have third-party network protocol stacks and services. Because the OS does not have any idea what kinds of errors these aftermarket services and stacks might be capable of generating, the Win32 error-reporting functions will just return `ERROR_EXTENDED_ERROR`. You can retrieve the provider-specific error information by using the `WNetGetLastError()` function:

```
Win32::NetResource::WNetGetLastError( $Error, $Description,
$Provider );
```

The values of the three parameters passed into the function are ignored, but they are assigned values. The parameters are set such that `$Error` will contain the error number, `$Description` will contain a textual description of the error, and `$Provider` will contain the name of the network provider that generated the error.

A simple way to collect `Win32::NetResource` error information would be to define a function, as in [Example 1.13](#).

### **Example 1.13 Retrieving a `Win32::NetResource` error**

```
01. use Win32::NetResource;
02. use Win32::WinError;
03.
04. sub NetError
05. {
06.     my( $Error, $Text, $Provider );
07.     $Error = Win32::GetLastError();
08.     if( ERROR_EXTENDED_ERROR == $Error )
09.     {
10.         Win32::NetResource::WNetGetLastError( $Error, $Text,
$Provider );
11.         $Result = "Error $Error: $Text (generated by $Provider)";
12.     }
13.     else
14.     {
15.         $Text = Win32::FormatMessage( $Error );
16.         $Result = "Error $Error: $Text";
17.     }
18.     return $Result;
19. }
```

The `NetError()` subroutine in lines 4 through 19 will return a string containing a description of the error.

There are three ways to retrieve errors when a `Win32::NetResource` function fails. You can use the following functions:

```
1. Win32::GetLastError();
2. Win32::NetResource::GetError( $Error );
3. Win32::NetResource::WNetGetLastError( $Error, $Text,
$Provider );
```

There has been much confusion about which function to use, so here is the scoop: When you execute a `Win32::NetResource` function and it fails, you should use `GetError()`:

```
Win32::NetResource::GetError( $Error );
```

This value that is set in `$Error` will be the error that was generated by the most recently called `Win32::NetResource` function. Using `Win32::NetResource::GetError()` is quite different from using `Win32::GetLastError()` in that you must pass a scalar into the `Win32::NetResource::GetError()` function, as in:

```
Win32::NetResource::GetError( $Error )
```

If the function successfully retrieves the error number, the scalar `$Error` will hold it.

### **Win32::RasAdmin Error Handling**

If a `Win32::RasAdmin` function fails, you can find the error it reported by using either the `Win32::RasAdmin::GetLastError()` or the `Win32::GetLastError()` function. Because RAS errors are not common to all Win32 platforms, the errors are not recognized by the traditional `Win32::FormatMessage()` function. Instead, you will need to use `Win32::RasAdmin::GetErrorString()` to find the textual message associated with the error number. The good news about this is that the error message will be correctly returned, regardless of whether the error is RAS related.

## **Summary**

The history of the various Win32 versions is long and twisted, with many interesting details. With all the changes made to Perl over the past few years to make it successfully run on the Win32 platform, it is no wonder that there are gaps in documentation and in the general understanding of the issues surrounding it.

One of the benefits of using Win32 Perl over other scripting utilities (such as batch files) is that the programmer is not limited to what the language is capable of. Perl has long had the capability to extend itself by making use of other languages such as C. By extending, Perl programmers are provided the promise of true Win32 integration.

Each discrete extension that Win32 Perl uses is, within itself, a library of functionality that some author found a need for. Although this distribution of work is good for variety, it also has resulted in many inefficient trends such as the lack of any conformity for important issues like how to retrieve error information.

Now that the basics have been covered, the rest of the standard Win32 extensions are yours for the using. You will find that with each extension, Perl is empowered with vast amounts of capabilities. Even those extensions that support only a couple of functions provide a Perl script with untold functionality.

Network administrators will find that Perl is a splendid tool for managing their network, but the extensions and functions described in the next chapter will make all the difference for them.

# **Chapter 2. Network Administration**

We often go about our day without ever thinking about networks. We pick up the phone and call someone next door, across the country, or on the other side of the globe. Those of us who play with

networks generally think about the network as a compilation of computers all talking to each other, but how often do we really consider the workings of our networks? Most of us don't think about it at all, or at least we try not to think about it.

Unlike most people, administrators have to constantly think about networks. Things like domain controllers, master browsers, shared resources, and dial-in connections can keep them up at night. Because it can be a nightmare to keep track of all of this, Win32 extensions have been written to make this easy to handle from Perl.

This chapter examines functions that manage the sharing of resources, resolve DNS names, discover network servers, and manage RAS servers.

The Win32 extensions discussed include the following:

- `Win32::AdminMisc`
- `Win32::LanMan`
- `Win32::NetAdmin`
- `Win32::NetResource`
- `Win32::RasAdmin`

## Discovering Servers on the Network

As a good administrator, you should be able to rattle off the names of your network servers without blinking an eye; however, a less fallible way to recall servers on the network would be to use the `Win32::NetAdmin::GetServers()` or `Win32::Lanman::NetServerEnum()` function. These functions, if successful, will populate an array with the name of each computer that matches the criteria that you supply.

Win32 platforms refer to computers by using their computer names. A *computer* name (also known as a NetBIOS or NetBEUI name) is unique to only one computer on a network, can be up to 15 characters in length, and is not case sensitive. Several of the Win32 extensions have functions that either accept or return a computer name. You need to be aware of the two forms of a computer name: a computer name and a proper computer name.

A computer name is just the name of the computer, as in, "My computer's name is '`DEVBOX1`'." A proper computer name, however, is the computer's name prefixed with two backslashes, as in `\DEVBOX1`.

This distinction is important because some functions handle one form but not the other. The `Win32::NetAdmin::GetServers()` function will return an array of computer names, for example, but the function `Win32::NetResource::NetShareGetInfo()` requires that you specify a proper computer name (if you specify a computer name at all).

### Note

*In the Win32 world, computers are most often grouped together in domains. These are not the same as Internet domains; they're more like workgroups. Domain names, like computer names, are not case sensitive and can be up to 15 characters in length. Unlike computer names, however, there is no such thing as a proper domain name. So prefixing a double backslash in front of a domain name does nothing.*

*In Windows NT networks, all domains must have what is known as a primary domain controller (PDC) that holds a database of all users and computers that participate in the domain. A PDC will manage all backup domain controllers (BDC) in the domain, performing some backup procedures over user database and other domain information. The PDC will occasionally update the BDCs with the latest information regarding the domain. This means that if an administrator changes something in a user account on a BDC, the PDC will eventually learn of this change and will then not only update itself but will also update all the other BDCs that exist in the domain.*

*When a computer or user logs on to a domain, either a PDC or a BDC will perform its authentication. Which machine performs the task depends on how busy they all are and how physically close the domain controller is to the requesting computer (typically, a domain controller, or DC, in the same subnet will respond before another DC that lives on a different subnet).*

*Many of the functions in Win32 extensions make reference to what is called a primary domain. When a computer or a user logs on to a domain, that domain becomes the primary domain—that is, the default domain. This is also known as the logon domain.*

*With the advent of the Active Directory, the domain model changes a bit with Windows 2000 and higher. There are no PDCs and BDCs. Instead, there are only domain controllers. Each DC has its own copy of the directory database (containing users, machines, groups, and so on). The DCs in the domain replicate this information to each other.*

*Changes to an account, a group, and so on are made to a domain controller. Eventually, the changes are propagated to the domain's Active Directory servers. These servers determine which modifications are the most recent and maintain them.*

Both the `GetServers()` and `NetServerEnum()` functions take four parameters:

```
Win32::NetAdmin::GetServers( $Machine, $Domain, $Flags, \@Array );
Win32::Lanman::NetServerEnum( $Machine, $Domain, $Flags, \@Array );
```

The first parameter (`$Machine`) is the name of the computer that you want to actually process the function. Usually, this is an empty string, and the local machine processes the request. It could, however, be any proper computer name on the network (such as `\server1`).

The second parameter (`$Domain`) is just the name of the domain on which you are requesting information. If this is empty, the primary domain of the machine specified in the first parameter is used.

The third parameter (`$Flags`) is a value that describes the type of machines you are looking for. You can use several constants in combination by logically OR'ing them together (see [Example 2.2](#)). [Table 2.1](#) contains the list of constants and their functionality. In older versions of these modules, not all of the constants are exposed, so you might need to specify the value instead of the constant name.

The fourth parameter for `GetServers()` (`\@Array`) is a reference to an array that will be populated with the machine names that fit the criteria specified in the `$Flags` parameter. If you are calling the `NetServerEnum()` function, the array is populated with an anonymous hash for each machine. The hash consists of the keys listed in [Table 2.2](#).

If the `GetServers()` or `NetServerEnum()` function is successful, it returns a `TRUE` value; otherwise, it returns a `FALSE` value.

**Table 2.1. Server Type Constants**

Constant	Value	Description
<code>SV_TYPE_AFP</code>	0x00000040	Servers running the Apple File Protocol.
<code>SV_TYPE_WORKSTATION</code>	0x00000001	Only NT workstations.
<code>SV_TYPE_SERVER</code>	0x00000002	Machines running the server service.
<code>SV_TYPE_SQLSERVER</code>	0x00000004	Only MS SQL Servers.
<code>SV_TYPE_DOMAIN_CTRL</code>	0x00000008	Primary domain controller.
<code>SV_TYPE_DOMAIN_BAKCTRL</code>	0x00000010	Backup domain controller.
<code>SV_TYPE_TIME_SOURCE</code>	0x00000020	Machines acting as a time server. <small>Win32::NetAdmin exposes this value as <code>SV_TYPE_TIMESOURCE</code>.</small>
<code>SV_TYPE_NOVELL</code>	0x00000080	Novell servers.
<code>SV_TYPE_DOMAIN_MEMBER</code>	0x00000100	LAN Manager 2.x domain members.
<code>SV_TYPE_PRINT</code> <code>SV_TYPE_PRINTQ_SERVER</code>	0x00000200	Machines sharing print queues.
<code>SV_TYPE_DIALIN</code> <code>SV_TYPE_DIALIN_SERVER</code>	0x00000400	Machines that have RAS services for remote dial-in.
<code>SV_TYPE_SERVER_UNIX</code> <code>SV_TYPE_XENIX_SERVER</code>	0x00000800	Xenix (UNIX) servers.
<code>SV_TYPE_NT</code>	0x00001000	Machines running NT (Workstation or Server).
<code>SV_TYPE_WFW</code>	0x00002000	Machines running Windows for Workgroups.
<code>SV_TYPE_SERVER_MFPN</code>	0x00004000	Microsoft's File and Print services for NetWare.
<code>SV_TYPE_SERVER_NT</code>	0x00008000	Windows NT/2000 workstations and services.
<code>SV_TYPE_POTENTIAL_BROWSER</code>	0x00010000	Machines able and willing to run the browser service.
<code>SV_TYPE_BACKUP_BROWSER</code>	0x00020000	Backup browsers for the domain.
<code>SV_TYPE_MASTER_BROWSER</code>	0x00040000	Master browser. (There can be a master browser for each subnet of a domain.)
<code>SV_TYPE_DOMAIN_MASTER</code>	0x00080000	Master browser for the entire domain.
<code>SV_TYPE_SERVER_OSF</code>	0x00100000	OSF-based servers. This is a legacy value and is rarely used.
<code>SV_TYPE_SERVER_VMS</code>	0x00200000	VMS-based servers. This is a legacy value and is rarely used.
<code>SV_TYPE_WINDOWS</code>	0x00400000	Windows 95 and higher.

**Table 2.1. Server Type Constants**

Constant	Value	Description
SV_TYPE_DFS	0x00800000	Machine is a Distributed file service (Dfs) tree root.
SV_TYPE_CLUSTER_NT	0x01000000	Windows NT/2000 server clusters available in the domain. <i>This constant is not exposed in Win32::NetAdmin. Use the value instead.</i>
SV_TYPE_TERMINALSERVER	0x02000000	Machines running Terminal Server services.  <i>This constant is not exposed in Win32::NetAdmin. Use the value instead.</i>
SV_TYPE_APPSERVER	0x10000000	Server running Citrix WinFrame.  <i>This constant is exposed in neither Win32::NetAdmin nor Win32::Lanman. Use the value instead.</i>
SV_TYPE_DCE	0x10000000	Machine is running IBM's Directory and Security Services (DSS) or some equivalent.  <i>This constant is not exposed in Win32::NetAdmin. Use the value instead.</i>
SV_TYPE_ALTERNATE_XPORT	0x20000000	Machines that have alternate file data transports, other than Server Message Blocks (SMB)/Common Internet File System (CIFS).
SV_TYPE_LOCAL_LIST_ONLY	0x40000000	Servers maintained by the browser on the local machine. This really only has useful meaning on master browsers.
SV_TYPE_DOMAIN_ENUM	0x80000000	List of domains.
SV_TYPE_ALL	0xFFFFFFFF	All members of the domain.

**Table 2.2. Win32::Lanman::NetServerEnum() hash keys**

Key	Description
comment	The machine's comment. This is a string that an administrator configured the machine to have.
name	The computer's name.
platform_id	This value indicates the type of platform the machine is running. See <a href="#">Table 2.3</a> for a list of valid platform ID values.
type	This value represents the types of services running on the machine. This is a bitmask consisting of any of the values in <a href="#">Table 2.1</a> .

**Table 2.2. Win32::Lanman::NetServerEnum() hash keys**

Key	Description
version_major	The major version number of the operating system. For example, if the OS is Windows NT 3.51, this value would be "3."
version_minor	The minor version number of the operating system. For example, if the OS is Windows NT 3.51, this value would be "51."

**Table 2.3. Platform IDs**

Constant	Value	Description
PLATFORM_ID_DOS	300 (0x012c)	The machine is running DOS. This includes machines running Windows 3.1x and Windows for Workgroups.
PLATFORM_ID_OS2	400 (0x0190)	The machine is running IBM's OS/2. This includes Windows 95/98/ME machines (which look similar to an OS/2 machine).
PLATFORM_ID_NT	500 (0x014f)	The machine is running Windows NT/2000.
PLATFORM_ID_OSF	600 (0x0258)	The machine is running OSF.
PLATFORM_ID_VMS	700 (0x02bc)	The machine is running VMS.

Suppose you want to get a list of all Windows machines in your domain. The code in [Example 2.1](#) would enable you to accomplish this task using `Win32::NetAdmin`.

[Example 2.2](#) is the same script using `Win32::Lanman`.

**Example 2.1 Retrieving a list of all the machines in a domain using Win32::NetAdmin**

```
01. use strict;
02. use Win32;
03. use Win32::NetAdmin;
04. my @List;
05. my $Domain = shift @ARGV || Win32::DomainName();
06. if( Win32::NetAdmin::GetServers( '', $Domain, SV_TYPE_ALL,
\@List ) )
07. {
08.   my $iCount;
09.   print "The machines in the $Domain domain are:\n";
10.  map
11.  {
12.    print ++$iCount, " ) $_ \n";
13.  } @List;
14. }
15. else
```

```

16. {
17. print Win32::FormatMessage( Win32::Lanman::GetLastError() );
18. }

```

**Example 2.2 Retrieving a list of all the machines in a domain using *Win32::Lanman***

```

01. use strict;
02. use Win32;
03. use Win32::Lanman;
04. my @List;
05. my $Domain = shift @ARGV || Win32::DomainName();
06. my %Platform = (
07.     300 => 'DOS',
08.     400 => 'OS/2, Windows 95/98/ME',
09.     500 => 'Windows NT/2000',
10.    600 => 'OSF',
11.    700 => 'VMS'
12. );
13. if( Win32::Lanman::NetServerEnum( '', $Domain, SV_TYPE_ALL,
14. \@List) )
14. {
15.     my $iCount;
16.     print "The machines in the $Domain domain are:\n";
17.     map
18.     {
19.         print ++$iCount, " ) $_->{name} ";
20.         print "($_->{comment}) " if( "" ne $$->{comment} );
21.         print $Platform{$_->{platform_id}};
22.         print " version:: $_->{version_major}. $_-
>{version_minor}\n";
23.     } @List;
24. }
25. else
26. {
27.     print Win32::FormatMessage( Win32::Lanman::GetLastError() );
28. }

```

In this case, the flag `$Type` is a bitmask consisting of `SV_TYPE_WINDOWS | SV_TYPE_WFW | SV_TYPE_NT`. This indicates that the names of all Windows-based computers in the primary domain will be returned.

[Example 2.3](#) assumes that you want the list of all domain controllers (primary and backup) and machines running Windows 95 (and later) for a domain passed in as a parameter. The machine `\server1` will produce the list. If no domain name is passed in, the script will assume that the default domain should be examined. This particular example makes passes in a hash reference instead of an array reference.

**Example 2.3 Retrieving a list of all primary and backup domain controllers in the domain**

```
01. use Win32;
```

```

02. use Win32::NetAdmin;
03. my %List;
04. my $Domain = shift @ARGV || Win32::DomainName();
05. my $Type = SV_TYPE_DOMAIN_BAKCTRL | SV_TYPE_DOMAIN_CTRL | 
SV_TYPE_WINDOWS;
06. if( Win32::NetAdmin::GetServers( '\\\\server1', $Domain, $Type,
%\List) )
07. {
08.     my $iCount;
09.     print "The machines in the $Domain domain are:\n";
10.    foreach my $Machine ( sort( keys( %List ) ) )
11.    {
12.        my $Comment = $List{$Machine};
13.        print ++$iCount, ") $Machine ";
14.        print "($Comment)" if( "" ne $$Comment );
15.        print "\n";
16.    }
17. }
18. else
19. {
20.     print Win32::FormatMessage( Win32::NetAdmin::GetError() );
21. }

```

## Note

*The core distribution's version of `Win32::NetAdmin::GetServers()` can accept a reference to a hash rather than an array for the fourth parameter. The hash is populated by keys with the machine name, and values are the machine's comment.*

## Tip

*In Perl, when you specify a backslash (\), you need to escape it with another backslash. So if you are specifying a server name that is \\Server, you will have to use double backslashes, as in:*

```
$Server = "\\\\"Server";
```

*This is why you will see double backslashes everywhere.*

*Even when using single quotation marks, you need to use double backslashes. This usually causes confusion for most coders because you do not have to typically escape a backslash when using single quotation marks. This is correct except that the task of specifying a proper computer name is one of those rare times when Perl, seeing your double backslash, will think you are escaping a backslash. Because of this, you must include two "escaped backslashes":*

```
$Server = '\\\\\"Server';
```

*Some Win32 extensions enable you to supply a server name using forward slashes (/), which the extension will flip automatically for you.*

Each element of the array will contain the name of one computer on the network that matches the criteria specified by the third parameter. The format of the computer is always just the name with no slashes. If you run the code in [Example 2.3](#), the output might resemble something like the following:

The machines in this domain are:

1. DEV
2. SERVER1
3. LIVERSAUSAGE

Notice how each element is only the name of a machine. (There are no prepended backslashes.)

You might notice that some functions enable you to specify a machine to perform the work (such as `Win32::NetAdmin::GetServers()` and `Win32::Lanman::NetServerEnum()`). If you are wondering why anyone would ever want to do this, you are in good company; most coders don't have a clue about this.

Usually, it doesn't matter which machine performs the task. Notice how I say "usually." In some instances, it is prudent to have another machine do the dirty work.

Assume, for example, that you want to get a list of all the current machines in the `ACCOUNTING` domain. You can use the following line:

```
Win32::NetAdmin::GetServers(' ', ' ', SV_TYPE_WORKSTATION, \@List);
```

This will have my machine figure out who is in my primary domain (`ROTH, NET`). If I want to discover the computer names for another domain, `ACCOUNTING`, however, I will have to ask a computer in that domain:

```
Win32::NetAdmin::GetServers('\\\Kyle', 'ACCOUNTING',
SV_TYPE_WORKSTATION, \@List);
```

If the primary domain for `\\\Kyle` is `ACCOUNTING`, I could have left the domain name empty.

## Finding Domain Controllers

I am always amazed how often I find code that assumes that a particular circumstance will be true. Someone will hardcode a script to refer to a primary domain controller (PDC), for example. It is possible for a primary domain controller to go down and a backup domain controller (BDC) to automatically step in and act as a temporary PDC, or an administrator can manually promote a BDC to take over the role of the PDC. (If this occurs, the PDC steps down and acts as a BDC.) Because of this, it is a bad idea to hardcode a PDC's computer name into a script. You should instead resolve the name of the domain's PDC using the `GetdomainController()` function:

```
Win32::NetAdmin::GetDomainController($Server, $Domain, $Name);
Win32::Lanman::NetGetDCName($Server, $Domain, $Name);
```

For this function, just like `GetServers()`, you specify the first parameter (`$Server`) as the name of the machine that will process the request (or use an empty string for the local machine). This parameter must be a proper machine name (the name is prepended with two backslashes). Typically, you will leave this entry as an empty string. If you specify a machine name, however, the function usually will execute more quickly.

The second parameter (`$Domain`) is the name of an NT domain or a blank string that represents the primary domain of the machine passed in as the first parameter.

The third parameter (`$Name`) is ignored when the function starts; if the function is successful, however, the name of the server is placed in it. Therefore, this value must be a scalar variable—you cannot use a constant value.

If `GetDomainController()` is successful, it will return a `true` value, and the variable passed in as the third parameter will be set to the name of the PDC. If the function fails, it will return a `false` value.

If you want to find the PDC for the primary domain, you could use the script from [Example 2.4](#).

## Tip

*The Win32::NetAdmin extension's `GetDomainController()` and `GetAnyDomainController()` functions take a scalar variable as the third parameter. This is unlike Win32::Lanman's similar functions, which take a reference to a scalar. This is important because it means that the scalar may have to be created before the function is called. In this example:*

```
use Win32::NetAdmin;
Win32::NetAdmin::GetDomainController( '', '', $Name);
```

*The scalar variable is created before being passed into the function. If it is successful, `$Name` will have a value of the domain's PDC. However, in the following code:*

```
use Win32::NetAdmin;
Win32::NetAdmin::GetDomainController( '', '', $Names{pdc});
```

*The `%Names` hash is created but the `pdc` key is not. Therefore, even though the function is successful, `$Names{pdc}` will not contain the PDC's name. This is corrected by first defining the hash's key, even if it defined with `undef`:*

```
use Win32::NetAdmin;
$Names{pdc} = undef;
Win32::NetAdmin::GetDomainController( '', '', $Names{pdc});
```

#### **Example 2.4 Discovering the primary domain controller for a domain**

```
01.use Win32::NetAdmin;
02.use Win32::Lanman;
03.my $PDC;
04.my $Domain = shift @ARGV || Win32::DomainName();
05.print "NetAdmin reports the PDC is: ";
06.if( Win32::NetAdmin::GetDomainController( '', $Domain, $PDC ) )
07.{  
08. print $PDC, "\n";
09.}  
10.else
11.{  
12. print Win32::FormatMessage( Win32::NetAdmin::GetError() );
13.}  
14.$PDC = undef;
15.print "Lanman reports the PDC is: ";
16.if( Win32::Lanman::NetGetDCName( '', $Domain, \$PDC ) )
17.{  
18. print $PDC, "\n";
19.}  
20.else
21.{  
22. print Win32::FormatMessage( Win32::Lanman::GetLastError() );
23.}
```

Notice how the `$PDC` name is prepended with backslashes (a proper Win32 machine name); this is different from the result you get when you use the `GetServers()` and `NetServerEnum()` functions.

#### **Tip**

*If you think that some of the extensions are a bit odd and not very consistent, you are catching on. Most of the Win32 extensions provide a direct interface to the Win32 API. This is fine for those who know the Win32 API and what to expect; for Perl programmers who have no knowledge of the idiosyncrasies that make up Win32, however, this will look very odd.*

*A perfect example is how some extension functions will return a machine name (without prepended backslashes) but expect an input parameter to be a proper computer name (with the prepended backslashes). Hopefully, future versions of extensions will be more consistent. Ideally, they would enable you to specify forward slashes rather than backslashes (keeping with how Win32 Perl enables you to specify either type of slashes in a file path).*

The `Win32::NetAdmin`'s `GetAnyDomainController()` function takes the same parameters as the `GetDomainController()` function. Instead of returning the primary domain controller, however, `GetAnyDomainController()` will return any domain controller. Usually the machine name returned will be the closest and least busy domain controller on your segment of the network. This is exactly the same as for the `Win32::Lanman`'s `NetGetAnyDCName()` function.

#### **Note**

The `Win32::NetAdmin::GetServers()` and `Win32::Lanman::NetServerEnum()` functions return an array of computer names. Conversely, the `GetAnyDomainController()`, `GetdomainController()`, `NetGetdCName()`, and `NetGetAnyDCName()` functions return a proper computer name (with prepended backslashes).

## Resolving DNS Names

Few things in life bring as much joy to a network administrator as being able to resolve DNS names to IP addresses. And, of course, Win32 Perl has many ways to resolve DNS names. You can use the `Net::DNS` module available from CPAN or Perl's built-in DNS functions such as `gethostbyname()` and its brethren. Win32 Perl did not always have such rich DNS support (early versions did not support the built-in functions). Because of this, the `Win32::AdminMisc` module included a few functions that are still available:

```
Win32::AdminMisc::GetHostAddress( $DNS_Name );
Win32::AdminMisc::gethostbyname( $DNS_Name );
Win32::AdminMisc::GetHostName( $IP_Address );
Win32::AdminMisc::gethostbyaddr( $IP_Address );
```

These four functions all work the same: If you pass in an IP address as the parameter, the DNS name will be looked up; if you pass in a DNS name as the parameter, the IP address will be looked up. The only reason there are multiple function names is for programmers who are accustomed to Perl's built-in `gethostbyXXX()` functions.

If these functions are successful, they will return either the IP address or the domain name (depending on what is passed into the function). If they fail, a `0` value is returned.

If these functions are called in an array context, an array of values is returned. Because many IP addresses can be mapped to a DNS name (and conversely many DNS names can be mapped to an IP address), calling one of these functions in an array context results in all names or IP addresses that match the requested resolution being discovered.

If you were to run the code in [Example 2.5](#), the IP address for `microsoft.com` would be printed.

### Example 2.5 Resolving a DNS name to an IP address

```
01. use Win32::AdminMisc;
02. my $DNSName = shift @ARGV || "microsoft.com";
03. print "The IP address is: ";
04. if( my $IP = Win32::AdminMisc::GetHostAddress( $DNSName ) )
05. {
06.   print "$IP\n";
07. }
08. else
09. {
10.   print "Unable to resolve the address.\n";
11. }
12. print "The list of IP addresses for $DNSName:\n";
13. if( my @List = Win32::AdminMisc::GetHostAddress( $DNSName ) )
14. {
```

```

15. map{ print "\t$_\n"; } @List;
16. }
17. else
18. {
19.   print "Unable to resolve the address.\n";
20. }

```

If you were to run the code in [Example 2.6](#), however, the DNS name would be printed. Notice that both [Example 2.4](#) and [Example 2.5](#) use the same function

`(Win32::AdminMisc::GetHostAddress())` even though they are looking for different results.

### **Example 2.6 Performing reverse DNS on an IP address**

```

01. use Win32::AdminMisc;
02. if( my $DNS =
Win32::AdminMisc::GetHostAddress( "192.168.1.1" ) )
03. {
04.   print "The DNS name is $DNS.\n";
05. }
06. else
07. {
08.   print "Unable to resolve the name.\n";
09. }

```

The code in [Example 2.7](#) would print out *all* the matching IP addresses for a given DNS name. Notice that line 5 calls the `GetHostAddress()` function in an array context.

### **Example 2.7 Discovering all IP addresses that match a DNS name**

```

01. use Win32::AdminMisc;
02. my $DNS = shift @ARGV || "microsoft.com";
03. my $iCount = 1;
04. print "The DNS name $DNS resolves to the following IP
addresses:\n";
05. foreach my $IP ( Win32::AdminMisc::GetHostAddress( $DNS ) )
06. {
07.   print $iCount++, " ") $IP\n";
08. }

```

#### **Tip**

*If your machine is configured for NetBEUI encapsulation over TCP/IP (NetBT), then under certain circumstances, performing a reverse DNS lookup will result in a NetBIOS network name instead of a DNS name. This is due to how the Windows DNS resolver code works. Therefore, instead of resolving an IP address to a DNS name of `server1.mydomain.com`, you might just get `server1`.*

*It is possible to prevent this type of resolution by disabling NetBIOS over TCP/IP. This is done by modifying the TCP/IP properties for a given network connection. However, doing this can have undesirable consequences because it would prevent other applications and services from resolving NetBIOS names as well.*

If your Perl script requires DNS resolution without NetBIOS names, use the `Net::DNS` module. Doing so will explicitly connect to DNS servers using the DNS protocol. This bypasses the Windows DNS resolver code that would perform any NetBIOS lookups.

Now, you might be wondering why you would use any of these rather than Perl's built-in DNS functions. Well, the answer is two-fold.

First, consider the ease of use. If you were to write some Perl code that resolves a DNS name to an IP address, it might look like [Example 2.8](#).

### **Example 2.8 Resolving a DNS name to an IP address using Perl's `gethostbyname()` function**

```
01. my $DNS = shift @ARGV || "www.roth.net";
02. if( my( $Addr ) = ( gethostbyname($DNS) )[4] )
03. {
04.     my @Quads = unpack('C4', $Addr);
05.     my $IP = join(".", @Quads);
06.     print "IP address for $DNS is $IP.\n";
07. }
08. else
09. {
10.     print "Failed to lookup $DNS.\n";
11. }
```

If you were to use the `Win32::AdminMisc` functions, it could look like [Example 2.9](#).

### **Example 2.9 Resolving a DNS name to an IP address using the `Win32::AdminMisc::GetHostAddress()` function**

```
01. use Win32::AdminMisc;
02. my $DNS = shift @ARGV || "www.roth.net";
03. if( my $IP = Win32::AdminMisc::GetHostAddress( $DNS ) )
04. {
05.     print "IP address for $DNS is $IP.\n";
06. }
07. else
08. {
09.     print "Failed to lookup $DNS.\n";
10. }
```

The difference between the Perl built-in function `gethostbyname()` and the `Win32::AdminMisc::GetHostAddress()` function is that `GetHostAddress()` takes care of unpacking the resulting data structure into a dotted quad address (lines 4 and 5 in [Example 2.8](#)).

The second reason for using the `Win32::AdminMisc` functions rather than the built-in Perl DNS resolution functions has to do with caching. The `AdminMisc` module caches the DNS names and IP addresses that it resolves.

Originally, the purpose of the `AdminMisc` DNS functions was to perform reverse DNS on IP addresses collected by an HTTP server for a Web statistics program. Caching was implemented to speed up the resolution of DNS names that were frequently looked up.

The `Win32::AdminMisc` DNS functions are not always the best way to go. If you have a need for portability between, let's say, UNIX, Macintosh, and Win32, using the `Win32::AdminMisc` extension will only defeat any advantage that it provides. All the DNS functions that `Win32::AdminMisc` provides can be simulated using Perl's native functions.

The caching of DNS entries can result in an increase in speed for scripts that continuously perform DNS lookups as well as reverse DNS resolutions. You can enable and disable caching with the `DNSCache()` function:

```
Win32::AdminMisc::DNSCache( [ $$State ] );
```

The only parameter (`$State`) is optional. If supplied, it sets the state of caching regardless of whether DNS caching is enabled. This value is either a `FALSE` (0) or `TRUE` (any nonzero) value.

If no parameter is passed in, the function will return the current state of DNS caching. If a parameter is specified, caching will be either activated (by passing in a `TRUE` value) or disabled (by passing in a `FALSE` value). By default, when the `Win32::AdminMisc` extension is loaded, caching is active.

The function will return a `TRUE` or `FALSE` value indicating the state of DNS caching. This return value reflects the state of caching after the function has been called.

You can determine how many DNS entries have been cached by using the `DNSCacheCount()` function:

```
Win32::AdminMisc::DNSCacheCount()
```

The function will return a value indicating how many DNS names have been cached. This number will not exceed the total number of cache elements (the value returned by `Win32::AdminMisc::DNSCacheSize()`). [Example 2.10](#) illustrates using the `DNSCacheCount()` function.

#### ***Example 2.10 Discovering the number of the DNS names that have been cached***

```
01. use Win32::AdminMisc;
02. my $TotalCached = Win32::AdminMisc::DNSCacheCount();
03. print "$TotalCached DNS entries have been cached.\n";
```

There are a finite number of entries that can be cached. This means that when the cache is full, some entries must be removed from cache so that new entries can be added. To help determine which entries are removed, a *least recently used* (LRU) cache algorithm is used. When the cache is full, the cache entry that has been least used recently (that is, that has been in cache the longest without being accessed) is replaced with the new DNS item.

By default, 600 cache entries are available in the cache. However, this size can be increased or decreased with the `DNSCacheSize()` function:

```
Win32::AdminMisc::DNSCacheSize( [ $Size ] )
```

The only parameter (`$Size`) is optional and represents the number of cache elements to allocate.

If a value is passed into the function, the DNS cache will be resized to the specified number of elements. This will reset the cache, resulting in the loss of any DNS names already cached.

## Tip

*There is a general misconception about the size of a cache. Many feel that the bigger the cache the better. In some cases this might be true, but typically too large a cache can impact the performance of a machine.*

*A cache works by looking for specific data in a database. If there are too many items in the database, the searching might take longer than it would to go out and fetch the data from the data source. Imagine if you set the DNS cache to 25,000 entries. If you look for a reference to [www.newriders.com](http://www.newriders.com) in the cache, it might take several seconds to search through all records, only to find that the entry is not in cache. However, to connect to a DNS server and look up the IP address might only take 5 seconds. In this particular case, it takes longer to search the cache than to query the DNS server.*

*Generally a sweet spot exists for each machine and network. You can determine the most optimized cache size (the sweet spot) by comparing how fast the computer is against the average time it takes to query a DNS server. This is a delicate value to determine; if your cache size is too small, you are wasting too much time always querying DNS servers and purging data from cache. If the value is too large, your machine is wasting time searching through the cache database.*

This function will return the number of elements that have been allocated for DNS caching. This value indicates the total number of DNS names that can be cached. [Example 2.11](#) demonstrates the use of `DNSCacheSize()`.

### **Example 2.11 Changing the size of the DNS cache**

```
01. use Win32::AdminMisc;
02. my $Total = Win32::AdminMisc::DNSCacheSize();
03. print "Total of $Total elements are allocated for DNS
caching.\n";
04. $Total = Win32::AdminMisc::DNSCacheSize( 1500 );
05. print "Now total of $Total elements are allocated for DNS
caching.\n";
```

## Note

*It is very important to note that when you change the cache size, all currently cached entries will be lost.*

## Case Study: Resolving DNS Names and IP Addresses

Suppose you need to write a script that will look up the DNS names for a list of IP addresses in a file. This file could be a list of IP addresses that have connected to your Web server, or it might be a list of addresses for your network's file servers. Assuming that this file contains only one IP address per line, you could use the script in [Example 2.12](#).

### **Example 2.12 Resolving DNS names and IP addresses**

```
01. use Win32::AdminMisc;
02. open( FILE, "< iplist.txt" ) || die "Could not open file
($!)\n";
03. # Set the DNS cache to 3500
04. Win32::AdminMisc::DNSCacheSize( 3500 );
05. while( my $Host = <FILE> )
06. {
07.     my( $DNS, $IP );
08.     # Remove any white space
09.     $Host =~ s/\s//gis;
10.    # Check if $Host is a DNS name or an IP address
11.    if( $Host =~ /[^\.\d]/ )
12.    {
13.        $IP = Win32::AdminMisc::GetHostName( $DNS = $Host );
14.    }
15.    else
16.    {
17.        $DNS = Win32::AdminMisc::GetHostName( $IP = $Host );
18.    }
19.    if( "" eq $$IP )
20.    {
21.        $IP = "UNKNOWN";
22.    }
23.    if( "" eq $$DNS )
24.    {
25.        $DNS = "UNKNOWN";
26.    }
27.    print "The DNS name for $IP is $DNS.\n";
28. }
29. close( FILE );
```

Let's assume that you create the text file **IPLIST.TXT** which contains the following text:

```
microsoft.com
www.dell.com
www.perl.org
198.41.0.6
198.137.240.92
208.202.218.15
www.newriders.com
```

When you run the Perl script in [Example 2.11](#), you will get the following output:

```
The DNS name for 207.46.230.218 is microsoft.com.  
The DNS name for 143.166.224.86 is www.dell.com.  
The DNS name for 166.84.0.8 is www.perl.org.  
The DNS name for 198.41.0.6 is rs.internic.net.  
The DNS name for 198.137.240.92 is www2.whitehouse.gov.  
The DNS name for 208.202.218.15 is www.amazon.com.  
The DNS name for 63.69.110.220 is www.newriders.com.
```

The sheer beauty of this code is that up to 3,500 DNS entries will be cached, so if the `IPLIST.TXT` file contains duplicate entries, there will be no need for your system to go out and requery a DNS server.

## Shared Network Resources

Part of the wonder of a network is that you can share information with others. This capability to share data and resources is what provides networks with their potential and their power. After all, what good would a network be without shared resources? It would be like a highway without on or off ramps, an airplane without an airport, a candidate without an election.

Sharing resources is such a prime function of a network that Win32 Perl has access to almost all the Win32 API's sharing functions. Pretty much any resource (directories and printers, for example) can be shared so that users on the network can connect to and use it.

You have several options when dealing with shared resources; you can create shares, delete shares, change the way resources are shared, connect to shares on remote machines, and disconnect from remote shares. All these functions are possible thanks to the `Win32::NetResource` and `Win32::Lanman` extensions.

Now, you might be wondering just what a share is. It is just as its name describes: a shared resource. If you share your C: drive so that users can access it over the network, you have created a share. Shares have names that are sometimes referred to as sharepoints. Such a name is very important because when others connect to your computer, they need to specify the name of the share to which they intend to connect. They do this by specifying a universal naming convention (UNC).

UNCs are like a path to a shared resource. You make a UNC by specifying the proper name of a computer (prepended with double backslashes) followed by a backslash and the name of the shared resource. If I were to try to connect to the UNC `\ServerA\TempFiles`, I would be trying to connect to a shared resource named `TempFiles` on my computer named `\ServerA`. (It just so happens that I have shared my `C:\Temp` directory with a name of `TempFiles` on that computer.)

### Tip

*Just like filenames on Win32 platforms, a share name can have spaces. For example*

`\ServerA\Program Files\`

*is a valid share name on the machine `\ServerA`.*

## Types of Network Resources

Resources come in two different types: a connectable share or a container. Connectable shares are just what the name implies: shares that can be connected to (such as printers and directories) and used by other users and machines. Containers (such as domains), unlike connectable shares, are resources that can hold other containers or multiple shares (such as networks and domains).

Consider when you share your `C:\TEMP` directory and name it `TempFiles`. This is a connectable share because remote users can connect to it and access the files in the `C:\TEMP` directory.

Containers, however, cannot be connected to; instead, they are just objects that hold other objects (such as containers and connectable shares). This is not as confusing as it might seem.

Suppose you double-click on the desktop's Network Neighborhood icon and see a window that looks like [Figure 2.1](#). There are two icons that represent two separate domains. These icons are containers; they are objects that contain other objects.

**Figure 2.1. An example of container objects (two domain icons).**



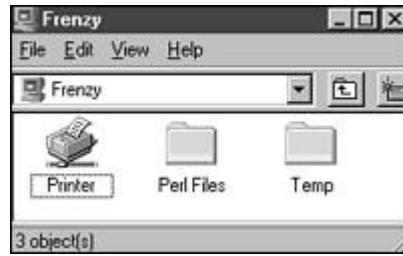
If you double-click on the `Roth.net` domain icon, you will see a set of computers (as in [Figure 2.2](#)). These icons each represent another container object.

**Figure 2.2. Another example of container objects (two computer icons).**



By double-clicking on the computer called `Frenzy`, you will see three shares, as illustrated in [Figure 2.3](#). These are connectable shares—that is, they are sharepoints that a user can actually connect to and use from across the network. As you can see, a container can hold connectable shares or containers.

**Figure 2.3. An example of connectable share objects (three shared resources).**



If we were to compare shares to hard drives, we could say that containers are directories (which can hold both files and directories), and connectable shares are files.

## Sharing Means Data Structures

Before learning how to deal with shares, you need to understand a few details. Resource sharing with Perl revolves around two data structures: the `NETrESOURCE` and `SHARE_INFO` hashes.

### **NETRESOURC E Hash Structure (Win32::NetResource)**

The `NETrESOURCE` hash is a data structure that represents a shared resource. It contains various attributes describing exactly how a resource is shared. These attributes are important, especially if a Perl script wants to connect to a remote shared directory over the network.

The `Win32::NetResource` and `Win32::Lanman` extensions use similar but different structures to share resources. This can lead to complications because a script cannot mix calls between these two extensions using the same hash.

First, let's look at `Win32::NetResource`'s `NETrESOURCE` hash. If you request a list of shared resources, you will retrieve an array of `NETrESOURCE` hashes like that shown in the following output (derived from using the `x` command in the debugger).

```
DB<1> x \%NetResource
0 HASH(0x1018b08)
  'Comment' => 'Okidata OL610e/PS PostScript Printer'
  'DisplayType' => 3
  'LocalName' => undef
  'Provider' => 'Microsoft Windows Network'
  'RemoteName' => '\\\\MAIN2\\\\PRINTER'
  'Scope' => 2
  'Type' => 2
  'Usage' => 1
```

The keys in a `NETrESOURCE` structure are as follows:

- `Comment`
- `DisplayType`
- `LocalName`
- `Provider`
- `RemoteName`
- `Scope`

- Type
- Usage

Each of these keys in the `NETRESOURCE` hash structure is described in more detail in the sections that follow.

### **The *Comment* Key**

The `Comment` key of the `NETRESOURCE` hash structure indicates a comment associated with the particular resource. For the output demonstrated previously, this would be '`Okidata OL610e/PS PostScript Printer`'.

### **The *DisplayType* Key**

The `DisplayType` key of the `NETRESOURCE` hash structure indicates how the object (either a connectable share or a container) should be displayed when you are browsing. Consider when you are using Explorer and you double-click on a computer in the Network Neighborhood. In the output to the list of shared resources, the `DisplayType` has a value of `3`, which corresponds to the `RESOURCEDISPLAYTYPE_SHARE` constant.

Possible values of this key are as follows:

- `RESOURCEDISPLAYTYPE_DOMAIN`. The object should be displayed as if it is a domain.
- `RESOURCEDISPLAYTYPE_GENERIC`. It does not matter how this specific type is displayed.
- `RESOURCEDISPLAYTYPE_FILE`. The object should be displayed as a file.
- `RESOURCEDISPLAYTYPE_GROUP`. The object should be displayed as a group.
- `RESOURCEDISPLAYTYPE_SERVER`. The object should be displayed as if it is a server.
- `RESOURCEDISPLAYTYPE_SHARE`. The object should be displayed as if it is a connectable share.
- `RESOURCEDISPLAYTYPE_TREE`. The object should be displayed as a tree (a hierarchical structure of some sort).

### **The *LocalName* Key**

This `LocalName` key of the `NETRESOURCE` hash structure refers to a local device if the `Scope` key is `RESOURCE_CONNECTED` or `RESOURCE_REMEMBERED`. Currently, the `Win32::NetResource` extension is hardcoded to always specify a scope of `RESOURCE_GLOBALNET`, however, so this key will always be undefined.

### **The *Provider* Key**

The `Provider` key of the `NETRESOURCE` hash structure tells which network provider handles this resource. Typically this will be Microsoft Windows Network, but it could be Novell NetWare or other network provider. If the provider is unknown, this string will be empty.

### **The *RemoteName* Key**

The `RemoteName` key of the `NETRESOURCE` hash structure is the name you use to reference this resource. This is usually a universal naming convention (UNC) or computer name such as `\server1\printer` or `\fileserver`. In a case in which the network resource is a container that you typically cannot directly access (such as a network provider, domain, or workgroup), the

`RemoteName` key will be the name of the object such as Microsoft Windows Network, Roth\_Domain, or Managers\_Group.

## The Scope Key

The `Scope` key of the `NETRESOURCE` hash structure represents which resources you are referring to. The possible values of this key are as follows:

- **RESOURCE\_CONNECTED**. All those resources to which your machine is currently connected. If this option is used, the `Usage` key will be undefined.
- **RESOURCE\_GLOBALNET**. All resources on the network (network providers, domains, workgroups, computers, printers, directories, and so on).
- **RESOURCE\_REMEMBERED**. Only those resources that are persistent (or remembered) connections—that is, connections that are automatically reconnected to when you log on to the machine. If this option is used, the `Usage` key will be undefined.

The trick about the `Scope` key is that it will always be `RESOURCE_GLOBALNET` (value of 2) because this is the value that is hardcoded into the extension. Refer to the description of the `LocalName` key for a brief explanation.

## The Type Key

The `Type` key of the `NETRESOURCE` hash structure refers to the type of resource the share is. The possible values of this key are as follows:

- **RESOURCETYPE\_ANY**. The share represents all resources. This is typically used when describing a domain (which is considered a share of multiple resources).
- **RESOURCETYPE\_DISK**. The resource is a directory.
- **RESOURCETYPE\_PRINT**. The resource is a printer.
- **RESOURCETYPE\_UNKNOWN**. The resource type is not known.

## The Usage Key

The `Usage` key of the `NETRESOURCE` hash structure specifies a bitmask that gives the resource usage. This member is defined only if `Scope` is `RESOURCE_GLOBALNET` (which is currently the only scope the extension supports). The possible values of this key are as follows:

- **RESOURCEUSAGE\_CONNECTABLE**. This resource type represents a connectable share; you can connect to this resource directly.
- **RESOURCEUSAGE\_CONTAINER**. This resource type represents a container resource; it can contain other resources. A domain object is a container, for example, because it can contain computer resources.

For anyone using `Win32::Lanman` instead of the `Win32::NetResource` extension, you will notice that its version of the `NETRESOURCE` hash uses different key names. Even though the functionality is identical, the names are different enough to break a script.

## The USE\_INFO hash (Win32::Lanman)

As mentioned in the preceding section, the `Win32::Lanman` extension does not really use the `NETRESOURCE` hash. Instead, it uses a similar structure known as `USE_INFO`. This hash consists of the keys described in the following sections.

### The *local* Key

This is the local name of a remotely shared resource. For example, in a connection to a remotely shared directory, the `local` key's value might be the drive letter to which it is mapped. For example, a remote shared directory might have a local key value of `d:`, and a remote shared printer might have a local key value of `lpt2:`.

This key's value can be undefined. This would indicate that no local device mapping is made to the remote resource. This is useful if you plan to access data from the shared resource using UNC's instead of drive letters. This way, the connection will always use any user credentials specified when the connection is initially made.

### The *remote* Key

The `remote` key refers to the remote shared resource. This is in the form of a UNC such as `\ServerA\ShareName`.

### The *password* Key

The `password` key refers to a password used to authenticate against the shared resource. For Windows 2000/NT machines, this password is used in conjunction with the `username` key. For DOS and Windows 95/98/ME shared resources, this key is the password for authenticating against the share itself regardless of username.

### The *statusKey*

The `status` key refers to the status of a shared resource. This key is not used with calls to the `NetUseAdd()` function.

Possible values are as follows:

Value	Meaning
<code>USE_OK</code>	The connection is successful.
<code>USE_PAUSED</code>	Paused by a local workstation.
<code>USE_SESSLOST</code>	Disconnected.
<code>USE_DISCONNECT</code>	An error occurred.
<code>USE_NETERR</code>	A network error occurred.
<code>USE_CONN</code>	The connection is being made.
<code>USE_RECONN</code>	Reconnecting.

### The *asg\_type* Key

The `asg_type` key refers to the type of resource is being accessed.

Possible values are as follows:

Value	Meaning
<code>USE_WILDCARD</code>	Matches the type of the server's shared resources. Wildcards can be used only with the <code>NetUseAdd()</code> function and only when the local member is not defined.
<code>USE_DISKDEV</code>	Disk device.
<code>USE_SPOOLDEV</code>	Spooled printer.
<code>USE_IPC</code>	Interprocess communication (IPC).

## The `refcount` Key

The `refcount` key refers to the number of objects open on the remote resource. For example, if five users have opened two files each through a shared directory, `refcount` for the that resource is 10.

This key is not used with the `NetUseAdd()` function.

## The `usecount` Key

The `usecount` key refers to number of connections attached to the shared resource.

This key is not used with the `NetUseAdd()` function.

## The `username` Key

The `username` key refers to a user who was specified when connecting to the shared resource.

## The `domainname` Key

The `domainname` key refers to the domain in which the shared resource exists. If this key is not defined or is an empty string (""), the current domain is used.

## The `SHARE_INFO` Structure

The `SHARE_INFO` hash is a data structure used when retrieving or setting information about a shared resource. This data structure is a hash that describes the attributes of the resource being shared.

From the Perl debugger, a dump of a `SHARE_INFO` hash (using the `x` command) might look like the following:

```
DB<1> x \%ShareInfo
0 HASH(0x106e138)
  'current-users' => 0
  'maxusers' => -1
  'netname' => 'Printer'
  'passwd' => ''
```

```
'path' => '\\\\MAIN2\\\\Printer,LocalsplOnly'  
'permissions' => 0  
'remark' => 'Okidata OL610e/PS PostScript Printer'  
'type' => 1
```

## Note

*Only members of the Administrators or Account Operators local group or those with Communication, Print, or Server operator group membership can successfully run these functions that accept or return the [SHARE\\_INFO](#) structure.*

The keys in a [SHARE\\_INFO](#) structure are as follows:

- `current_uses`
- `max_uses`
- `netname`
- `passwd`
- `path`
- `permissions`
- `remark`
- `security_descriptor`
- `type`

The keys in the [SHARE\\_INFO](#) hash structure are described in more detail in the sections that follow.

### The `current_uses` Key

The `current_uses` key of the [SHARE\\_INFO](#) hash structure refers to the number of users currently connected to the resource. The [Win32::NetResource](#) extension incorrectly refers to this key as "current-users".

### The `max_uses` Key

The `max_uses` key of the [SHARE\\_INFO](#) hash structure refers to the number of users who can access the resource simultaneously. If no limit is imposed, the value will be `-1`. The [Win32::NetResource](#) extension incorrectly refers to this key as "maxusers".

### The `netname` Key

The `netname` key of the [SHARE\\_INFO](#) hash structure refers to the name of the shared resource. This is the name that the resource was shared as. If a sharepoint is accessed by using the UNC `\\\\server1\\\\WordFiles`, for example, the `netname` is `WordFiles`.

### The `passwd` Key

If the server uses share-level security (as Windows 95 does), the `passwd` key of the [SHARE\\_INFO](#) hash structure is the password for the share. This password cannot be longer than eight characters.

If the server uses user-level security (as Windows NT/2000 does), this key is ignored.

## The *path* Key

The *path* key of the `SHARE_INFO` hash structure refers to the path that the server is sharing. If a server shares the directory `C:\TEMP` with a name of `Temp File`, for example, the path would be `C:\TEMP`.

## The *permissions* Key

The *permissions* key of the `SHARE_INFO` hash structure contains a value that represents permissions placed on the shared resource. These permissions restrict how connected users can access the resource.

If the server sharing the resources makes use of user-level security, this key is ignored and is assigned a value of `0`. This is the case for Windows NT machines but not Windows 95/98/ME. The capability to specify user-level permissions (that is, permissions assigned to users and/or groups) on a share is currently not supported by any of the standard Win32 extensions.

The value of this key is a bitmask comprising the values outlined in [Table 2.4](#).

**Table 2.4. List of Constants Used with the Permissions Key in a `SHARED_INFO` Hash**

Constant	Value	Description
<code>ACCESS_NONE</code>	( <code>0x00</code> )	No permissions are granted.
<code>ACCESS_READ</code>	( <code>0x01</code> )	Permission to read data from a resource and, by default, to execute the resource.
<code>ACCESS_WRITE</code>	( <code>0x02</code> )	Permission to write data to the resource.
<code>ACCESS_CREATE</code>	( <code>0x04</code> )	Permission to create an instance of the resource (such as a file); data can be written to the resource as the resource is created.
<code>ACCESS_EXEC</code>	( <code>0x08</code> )	Permission to execute the resource.
<code>ACCESS_DELETE</code>	( <code>0x10</code> )	Permission to delete the resource.
<code>ACCESS_ATTRIB</code>	( <code>0x20</code> )	Permission to modify the resource's attributes (such as the date and time when a file was last modified), execute, and delete resources.
<code>ACCESS_PERM</code>	( <code>0x40</code> )	Permission to modify the permissions (read, write, create, execute, and delete) assigned to a resource for a use or application.
<code>ACCESS_ALL</code>	( <code>0x7f</code> )	Permission to read, write, create, execute, and delete resources and to modify their attributes and permissions.

## Note

*For some reason, the permission constant names were not coded into older versions of the Win32::NetResource extension; therefore, you must use the numeric values rather than the constant names.*

*If you are creating a `SHARED_INFO` hash (`$Info`) and want to specify the read-only permission, you must use the value `0x01` rather than the constant name `ACCESS_READ`, as in:*

```
$Info{ 'permission' } = 0x01;
```

*This might change in future versions of the `Win32::NetResource` extension.*

If you want to specify a value comprising several attributes, you need to logically OR (|) them together. If you want to specify read and write permissions, you use the following:

```
$Permission = 0x01 | 0x02;
```

Conversely, if you want to test and see whether a particular permission is set, you logically AND (&) the permission with the attribute. If you have a particular permission, for example, you can test whether it has write permissions by using the following:

```
$CanWrite = ($Permission & 0x02) != 0;
```

The variable `$CanWrite` will result in either a 0 if the attribute is not present or a 1 if it is present.

### **The `remark` Key**

The `remark` key of the `SHARE_INFO` hash structure refers to the comment assigned to the shared resource.

### **The `security_descriptor` Key**

The `security_descriptor` key consists of a proper Win32 security descriptor. Such a security descriptor can be created and analyzed by the `Win32::Perms` extension. This key determines which users and groups have what kind of access. It also determines which users and groups are audited so that logs are made for accessing the shared resource. See [Chapter 11](#), "Security," for more details on security descriptors and `Win32::Perms`.

This key is only supported by the `Win32::Lanman` extension.

### **The `type` Key**

The `type` key of the `SHARE_INFO` hash structure refers to what type of resource the share is. Possible values of this key are as follows:

- `0x00`. A shared directory
- `0x01`. A shared printer queue

- **0x02**. A shared communication device (such as a modem)
- **0x03**. Interprocess communication (IPC)

## Connecting to Shared Network Resources

If your intention is to make a connection to a network share, you must visit the `Win32::NetResource::AddConnection()` and `Win32::Lanman::NetUseAdd()` functions:

```
Win32::NetResource::AddConnection( \%NetResource, $Password,
$Userid, $Connection );
Win32::Lanman::NetUseAdd( \%NetResource );
```

The first parameter is a reference to a `NETRESOURCE` hash. The `NetUseAdd()` function only takes this parameter.

The second and third parameters (`$Password` and `$Userid`) are the password and user ID, respectively. The user ID can be any valid username. (It does not have to be the current user.) If the `$Userid` parameter is an empty string, the current user's user ID will be used.

The fourth parameter (`$Connection`) determines whether the system should remember this connection. If the value for this parameter is a `1`, the connection will be remembered, and the next time the user logs on, there will be an attempt to connect again (the connection will be persistent). This information will be remembered, however, only if it is successful in connecting to the share and the connection redirects a local device. Deviceless connections (that is, connecting as a UNC and not as a drive or printer) will not be remembered.

If the `AddConnection()` function is successful, it returns a `true` value; otherwise, it returns `false`.

The code in [Example 2.13](#) enables you to connect your `R:` drive to `\server1\games` and remember the connection; therefore, every time you log on, your `R:` drive will automatically connect.

The code in [Example 2.13](#) will connect the `R:` drive to `\server1\games`. Notice that only two fields of the `NETRESOURCE` hash were used: `LocalName` and `RemoteName`. There really is no need to use the other fields. If you wanted to specify a user ID and password, you would fill out both the `Password` and `User` parameters. Typically, these parameters are left as empty strings so that the connection is made using the current user's ID and password. The fourth parameter of `AddConnection()` in [Example 2.12](#) is a `1`. So if the function successfully connects, the user's profile is updated to reconnect every time the user logs on.

### Note

*In [Example 2.13](#), you could have left off the `LocalName` key from the `%Netresource` hash. This would have made a deviceless connection. You would have been connected to the shared resource but not through a local drive letter. The only way to access the resource would be through a UNC.*

The user's profile would not have been updated to remember the connection (because deviceless connections cannot be persistent, whereas regular local devices connected to a share can be).

### **Example 2.13 Connecting to a shared resource using Win32::NetResource**

```
01. use Win32::NetResource;
02. my %NetResource = (
03.     LocalName => "R:",
04.     RemoteName => "\\\ServerA\\Files\$"
05. );
06. my $User = "";
07. my $Password = "";
08. if( Win32::NetResource::AddConnection( \%NetResource,
$Password, $User, 1 ) )
09. {
10.     print "Successful!\n";
11. }
12. else
13. {
14.     # See Example 1.13 for the NetError() function
15.     print NetError();
16. }
```

If you are using the `Win32::Lanman::NetUseAdd()` function, there are a few things to consider. You will be using the `USE_INFO` hash. The only key that needs to be specified is `remote`. This key identifies the remote shared resource.

Example 2.14 specifies a `local` key on line 3. This will map the `R:` drive to the remote share. If this key was not specified, then a connection would be made to the shared resource, but there would be no local drive mapping. This would result in a deviceless connection.

The example also identifies what type of resource to connect to (line 5). This argument is not necessary because Win32 is smart enough to figure out what type of resource you are connecting.

Notice that there is no `username` or `password` specified. If no `username` key is defined, the local user is assumed. If the key is defined with `undef` or an empty string, a user name of "" is attempted to connect to the resource. This will most likely fail to authenticate. A `password` can be defined even though the `username` key is not defined. This is useful when connecting to a DOS or Windows 95/98/ME shared directory. Such shares can be password protected and do not consider usernames.

### **Example 2.14 Connecting to a shared resource using Win32::Lanman**

```
01. use Win32::Lanman;
02. my %UseInfo = (
03.     local => "R:",
04.     remote => "\\\ServerA\\Files\$",
05.     asg_type => USE_DISKDEV
06. );
07. if( Win32::Lanman::NetUseAdd( \%UseInfo ) )
08. {
09.     print "Successful!\n";
10. }
11. else
12. {
```

```
13.     print  
Win32::FormatMessage( Win32::Lanman::GetLastError() );  
14. }
```

## Finding Information About Shared Network Resources

When you share many resources, it can be easy to forget the details of a share. This is when `NetShareGetInfo()` is used:

```
Win32::NetResource::NetShareGetInfo( $ShareName, $ShareInfo [,  
$Machine]);  
Win32::Lanman::NetShareGetInfo( $Machine, $ShareName,  
\%ShareInfo );
```

This function is found in both `Win32::NetResource` and `Win32::Lanman` extensions. There are four differences between these two functions:

1. The `Win32::NetResource::NetShareGetInfo()` function's machine name is optional.
2. The `Win32::NetResource::NetShareGetInfo()` function *must* pass the hash structure as a scalar—not as a hash reference.
3. The order of parameters is different between the two functions.
4. The resulting hash has some slight differences, as described in the earlier section "The `SHARE_INFO` Structure."

The `$ShareName` parameter is the name of the sharepoint (like `TempFiles` from the UNC `\server1\TempFiles`).

The `$ShareInfo` and `\%ShareInfo` parameters are references to a hash that will be filled in with a `SHARE_INFO` structure. The `Win32::NetResource::NetShareGetInfo()` function only accepts a scalar, not a hash or a hash reference! *If you use a reference to a hash (`\%ShareInfo`), the resulting hash will be empty.* You must use a scalar and dereference the hash to access the hash's keys, as in [Example 2.15](#).

The `$Machine` parameter is optional and represents the name of the computer that will perform the lookup. So if you want to get information about a share on another machine, you would specify the proper machine name (prepended with double backslashes) as the third parameter. If this parameter is an empty string or is left out of the function, the local computer is assumed.

If the `NetShareGetInfo()` function is successful, it returns a `TRUE` value; otherwise, it returns `FALSE`.

In [Example 2.15](#), you can see how many users are connected to a particular share by using this function. The example calls the function from both extensions to illustrate their slight differences.

### Note

*As with any remote administration function, the user running the script must have appropriate permissions on the remote machine.*

### **Example 2.15 Retrieving information about a shared resource**

```
01. use Win32::NetResource;
02. use Win32::Lanman;
03. my $Machine = shift @ARGV || "\\\server";
04. my $ShareName = @ARGV || "share";
05. my $Info = {};
06. print "NetResource reports:\n";
07. if( Win32::NetResource::NetShareGetInfo( $ShareName, $Info,
$Machine ) )
08. {
09.     print "The share $Machine\\$ShareName ";
10.    # Notice that Win32::NetResource incorrectly references the
11.    # 'current-uses' key as 'current-users'.
12.    print "has $Info->{'current-users'} current connections.\n";
13. }
14. else
15. {
16.     my $NetError;
17.     Win32::NetResource::GetError($NetError);
18.     print Win32::FormatMessage($NetError);
19. }
20. my %Info;
21. print "Lanman reports:\n";
22. if( Win32::Lanman::NetShareGetInfo( $Machine, $ShareName,
%\Info ) )
23. {
24.     print "The share $Machine\\$ShareName ";
25.     print "has $Info{'current_uses'} current connections.\n";
26. }
27. else
28. {
29.     print Win32::FormatMessage( Win32::Lanman::GetLastError() );
30. }
```

## **Changing Shared Network Resource Attributes**

Just as you have the capability to change your mind, you have the capability to change attributes of an existing share. You might want to change the number of concurrent connections a share can have or maybe change the comment field of the share. All this is made possible with a function called `NetShareSetInfo()`:

```
Win32::NetResource::NetShareSetInfo( $ShareName, \%ShareInfo,
$Param [, $Machine] );
Win32::Lanman::NetShareSetInfo( $Machine, $ShareName,
%\ShareInfo );
```

### **Note**

*Due to a bug in the `Win32::NetResource::NetShareSetInfo()` function, you are advised to use the `Win32::Lanman` version instead. In particular, the `Win32::NetResource` version of the function does not set a security descriptor, nor does it set the descriptor to `NULL`. This almost always results in the function failing, setting the `$Param` parameter with a packed value of `501`. See the "Creating Shared Network Resources" section later in this chapter for a discussion on this parameter.*

Similar to the `NetShareGetInfo()` function, the `$ShareName` parameter is the name of the sharepoint.

The `\%ShareInfo` parameter is a hash reference to a `SHARE_INFO` hash. In the `Win32::NetResource` version of this function, this parameter can be a hash reference, unlike its brother `Win32::NetResource::NetShareGetInfo()`, which *must* be a scalar variable. The `Win32::Lanman` version does not have these limitations.

The `$Machine` parameter represents the name of the computer that will perform the lookup. So if you want to get information about a share on another machine, you would specify the proper machine name (prepended with double backslashes) as this parameter. If this is an empty string, the local computer is assumed. This parameter is optional when using the `Win32::NetResource` version of the function.

The `$Param` parameter must be a scalar because it will contain a returned value. The value is only used for debugging purposes and is of little use. This parameter *must* be defined before it is passed into the function. Usually, you would simply want to assign a `0` value to the `$Param` variable before calling the function. For more information, see the `$Param` parameter in the "Creating Shared Network Resources" section. This value is not used in the `Win32::Lanman` extension.

If the `NetShareSetInfo()` function is successful, it will return a `TRUE` value; otherwise, it returns `FALSE`.

## Tip

*There is a trick to using the `NetShareSetInfo()` function. If you don't fill out the `SHARE_INFO` hash fully (leaving out the `maxusers` key, for example), you run the risk of setting values to null. If you filled out the hash to look like the following, for example:*

```
$ShareInfo = {
    'netname' => 'Graphic',
    'passwd' => '',
    'path' => 'C:\\Data\\\\Graphics',
    'permissions' => 0,
    'remark' => 'Come and get my graphics!',
    'type' => 0;
}
```

*No one will be able to connect to your share. Notice that the `maxusers` key (or the `max_use` key if you are using `Win32::Lanman`) was not specified. When you call `NetShareSetInfo()`, the `maxusers` parameter will be set to `0` because it was not specified. This setting will cause your shared resource to allow zero users to connect to it at any time. This is not good.*

The best way to call the `NetShareSetInfo()` function is to first call `NetShareGetInfo()`. When done, change the resulting hash values to your liking and then call `NetShareSetInfo()` (refer the Tip sidebar for details). [Example 2.16](#) does exactly this. Line 7 gets the info hash by calling `NetShareGetInfo()`. Line 13 changes the `remark` in the hash, and then line 14 updates the hash information.

To run this script, just pass in the UNC to the share you want to modify followed by the new comment. For example:

```
perl Example 2.16.pl \\myserver\myshare "My New Comment"
```

### ***Example 2.16 Changing the remark field of a shared resource***

```
01. # Use Win32::Lanman instead of Win32::NetResource because
02. # of a security descriptor bug in
03. # Win32::NetResource::NetShareSetInfo()
04. use Win32::Lanman;
05. my( $Machine, $ShareName ) = ( shift @ARGV || die ) =~
/^(\\\\.+?)\\(.*)$/;
06. my $NewRemark = shift @ARGV || die;
07. if( ! Win32::Lanman::NetShareGetInfo( $Machine, $ShareName,
%\%Info ) )
08. {
09.     print "Unable to retrieve share information.\n";
10.    PrintError();
11.    exit;
12. }
13. $Info{remark} = $NewRemark;
14. if( Win32::Lanman::NetShareSetInfo( $Machine, $ShareName,
%\%Info ) )
15. {
16.     print "The Remark has been successfully changed.\n";
17. }
18. else
19. {
20.     print "Unable to set the new remark.\n";
21.     my $Error = Win32::Lanman::GetLastError();
22.     if( ERROR_EXTENDED_ERROR == $Error )
23.     {
24.         my( $Text, $Provider, $Result );
25.         Win32::NetResource::WNetGetLastError( $Error, $Text,
$Provider );
26.         $Result = "Error $Error: $Text (generated by $Provider)";
27.     }
28.     else
29.     {
30.         $Text = Win32::FormatMessage( $Error );
31.         $Result = "Error $Error: $Text";
32.     }
33.     print $Result, "\n";
```

```
34. }
```

Another example of modifying a network share is illustrated in [Example 2.17](#). Here we are removing any permissions that might have been granted to the Guest account and the Domain Guests and Everyone groups. This is accomplished by using the `Win32::Perms` extension to modify the security descriptor that `Win32::Lanman::NetShareGetInfo()` returns in the `%Info` hash. Mind you, this is only possible using `Win32::Lanman` because `Win32::NetResource` neither returns nor sets the security descriptor on a share.

In line 4, the script populates the `%Info` hash with the share's configuration. Line 10 creates an empty new `Win32::Perms` object, and line 11 imports all of the security permissions from the share's security descriptor. Lines 12 through 14 remove all references to the Guest account as well as the Everyone and Domain Guests groups.

In line 15, the `Win32::Perms` object exports the security descriptor in what is called *self-relative* format. This is the format required when setting a security descriptor on a network share. The exported security descriptor is then placed in the `%Info` hash. Finally, line 16 sets the new configuration information on the network share.

Actually the same operation could have been carried out using only `Win32::Perms`. See [Chapter 11](#) for more details on using this extension.

To run this script, just pass in the UNC to the share you want to modify, such as:

```
perl Example 2.17.pl \\myserver\myshare
```

### **Example 2.17 Changing the permissions on a network share**

```
01. use Win32::Lanman;
02. use Win32::Perms;
03. my( $Machine, $ShareName ) = ( shift @ARGV || die ) =~
/^(\\\.\+?)\\(.*)$/;
04. if( ! Win32::Lanman::NetShareGetInfo( $Machine, $ShareName,
%\Info ) )
05. {
06.   print "Unable to retrieve share information.\n";
07.   PrintError();
08.   exit;
09. }
10. my $PermsObj = new Win32::Perms() || die;
11. $PermsObj->Import( $Info{security_descriptor} );
12. $PermsObj->Remove( "Guest" );
13. $PermsObj->Remove( "Domain Guests" );
14. $PermsObj->Remove( "Everyone" );
15. $Info{security_descriptor} = $PermsObj->GetSD( SD_RELATIVE );
16. if( Win32::Lanman::NetShareSetInfo( $Machine, $ShareName,
%\Info ) )
17. {
18.   print "The permissions have been successfully changed.\n";
19. }
```

```

20. else
21. {
22.     print "Unable to set the new remark.\n";
23.     my $Error = Win32::Lanman::GetLastError();
24.     print "Error: ", Win32::FormatMessage( $Error );
25. }
26. $PermsObj->Close();

```

## Note

*You must be a member of the Administrators or Server Operators group to successfully use the `Win32::NetResource::NetShareSetInfo()` and `Win32::Lanman::NetSharegetInfo()` functions; however, members of the Print Operators group can use this function (but only on shared printer resources).*

## Disconnecting Connections to Shared Network Resources

To cancel (or disconnect) from a shared resource (for example, a mapped drive), you use the `Win32::NetResource::CancelConnection()` function or the `Win32::Lanman::NetUseDel()` function:

```

Win32::NetResource::CancelConnection( $LocalName, $Flag, $Force );
Win32::Lanman::NetUseDel( $LocalName [, $Force] );

```

The `$LocalName` parameter is the local name of the shared resource. If you have connected to `\server1\TempFiles` as your `R:` drive, for example, the local name is `R:`. If you have mapped your `LPT2:` port to some printer share, your local name is `LPT2:`. If you have connected to `\server1\source` but have not mapped it to any local device, however, you would use the full UNC as the local name.

The `$Flag` parameter is a flag that indicates how you want to cancel the connection. If you specify a `0`, the connection is terminated. If you specify a `1`, however, not only will the connection be canceled but also the user's profile will be updated, canceling any persistent connection to the resource. Use this if you want to prevent Windows from automatically connecting to the share the next time you log on.

The `$Force` parameter specifies whether force should be used to disconnect. This value of this parameter differs depending on what function is called.

If you are using `Win32::NetResource::CancelConnection()`, the `$Force` parameter is a Boolean value. A value of `1` means that the connection will be forced to disconnect even if there are open files or applications running from that resource. If a value of `0` is specified and there are open files, the function will fail.

If you are using `Win32::Lanman::NetUseDel()`, the `$Force` parameter can have any one value from [Table 2.5](#). Note that the official MSDN documentation for this function does not really indicate any difference between the `USE_NOFORCE` and `USE_FORCE` values.

If either the `CancelConnection()` or `NetUseDel()` function is successful, it returns `true`; otherwise, it returns `false`.

**Table 2.5. Disconnect Values**

Constant	Value	Description
<code>USE_NOFORCE</code>	0	Fail the disconnection if open files exist on the connection. There is no known difference between this value and <code>USE_FORCE</code> .
<code>USE_FORCE</code>	1	Fail the disconnection if open files exist on the connection. There is no known difference between this value and <code>USE_NOFORCE</code> .
<code>USE_LOTS_OF_FORCE</code>	2	Close any open files and delete the connection.

Let's say one day your boss demands that you disconnect all of your connections to remote drives and remove all persistent connections to them. You can use the code in [Example 2.18](#).

### **Example 2.18 Disconnecting your machine from all shared disk resources**

```
01. use Win32::NetResource;
02. use Win32::AdminMisc;
03. my @RemoteDrives = Win32::AdminMisc::GetDrives( DRIVE_REMOTE );
04. foreach my $Drive ( @RemoteDrives )
05. {
06.     Win32::NetResource::CancelConnection( $Drive, 1, 1 );
07. }
```

This example makes use of the `Win32::AdminMisc::GetDrives()` function, which returns an array of valid drive root directories that match the specified drive type. For more details, see [Chapter 3](#), "Administration of Machines."

## **Creating Shared Network Resources**

Creating a share is as easy as filling out a data structure and calling a function. Just fill out a `SHARE_INFO` hash (refer to the section "The `SHARE_INFO` Structure" earlier in this chapter) and then submit a reference to that structure to either the `Win32::NetResource::NetShareAdd()` or `Win32::Lanman::NetShareAdd()` function:

```
Win32::NetResource::NetShareAdd( $Share_Info_Ref, $Param [ ,
$Machine ] );
Win32::Lanman::NetShareAdd( $Machine, \%Info );
```

The `$Share_Info_Ref` parameter passed into `Win32::NetResource::NetShareAdd()` *must* be a reference to a `SHARE_INFO` hash, not just a hash. You can use a reference to a hash:

```
%Share = (
    'path' => "C:\\Temp",
    'maxusers' => -1,
```

```

'netname' => "TempFiles",
'remark' => "My Temporary Files",
'passwd' => "",
'permissions' => 0,
);
Win32::NetResource::NetShareAdd( \%Share, $Param );

```

Or you can use an anonymous hash:

```

$Share = {
    'path' => "C:\\Temp",
    'maxusers' => -1,
    'netname' => "TempFiles",
    'remark' => "My Temporary Files",
    'passwd' => "",
    'permissions' => 0,
};
Win32::NetResource::NetShareAdd( $Share, $Param );

```

Or even:

```

Win32::NetResource::NetShareAdd( {
    'path' => "C:\\Temp",
    'maxusers' => -1,
    'netname' => "TempFiles",
    'remark' => "My Temporary Files",
    'passwd' => "",
    'permissions' => 0,
}, $Param );

```

Any of the preceding reference methods will work just fine.

The `$Param` parameter is a scalar that will be assigned an error value. This error value is only of use if the function fails and the error is `ERROR_INVALID_PARAMETER`. (This constant comes from the `Win32::WinError` extension.) If both of these conditions are true, the `$Param` variable is set to be the index of the parameter that caused the failure. The value of `$Param` is used internally and is of very little use to a Perl program. You should just ignore this parameter. However, in case you are curious... the value of `$Param` is a binary DWORD (an unsigned long integer). You can convert the parameter into a valid value using:

```
$Value = unpack( "S", $Param );
```

Older versions of this extension populate the `$Param` variable with the value already unpacked. The value indicates the problem with the function call. The possible values are displayed in [Table 2.6](#).

**Table 2.6. List of Possible NetShare Error Parameter Values**

Constant	Value	Description
SHARE_NETNAME_PARMNUM	0x01	The network name is in error.
SHARE_TYPE_PARMNUM	0x03	The type is not valid.
SHARE_REMARK_PARMNUM	0x04	The remark field is invalid.
SHARE_PERMISSIONS_PARMNUM	0x05	The permissions field is invalid.
SHARE_MAXUSES_PARMNUM	0x06	The max_uses (max-users) field is invalid.
SHARE_CURRENTUSES_PARMNUM	0x07	The current_uses (current-users) field is invalid.
SHARE_PATH_PARMNUM	0x08	The path field is invalid.
SHARE_PASSWD_PARMNUM	0x09	The password is invalid.
SHAREFILE_SD_PARMNUM	0x01f5 (501 decimal)	The security descriptor is invalid. See <a href="#">Chapter 11</a> for details on security descriptors.

The \$Machine parameter is the name of a computer on which you want the share to be created. This is optional in the `Win32::NetResource` version of the function. If you do not include this parameter, the share will be created on the local machine. Leaving off the parameter is the same as specifying an empty string as the machine name. The valid format for the machine name is either the computer's name (`server1`) or the proper computer name prepended by two backslashes (`\server1`).

The \$Machine parameter must be provided for the `Win32::Lanman` version of the function. However, the parameter can be an empty string (""), indicating to use the local machine.

If the `NetShareAdd()` function is successful, it returns a `TRUE` value; otherwise, it returns `FALSE`. [Example 2.19](#) demonstrates how to share a directory.

### ***Example 2.19 Creating a shared directory***

```

01. use Win32::NetResource;
02. use Win32::WinError;
03. my %Share = (
04.   'path' => "C:\\Temp",
05.   'maxusers' => -1,
06.   'netname' => "TempFiles",
07.   'remark' => "My Temporary Files",
08.   'passwd' => "",
09.   'permissions' => 0,
10. );
11. my $Machine = "\\\server1";
12. if( Win32::NetResource::NetShareAdd( \%Share, $Param,
$Machine ) )
13. {
14.   print "Successfully shared $Share{path} as
$Share{netname}.\\n";
15. }
```

```

16. else
17. {
18.     print "Failed to share $Share{path} as $Share{netname}.\\n";
19.     Win32::NetResource::GetError( $Error );
20.     if( $Error == ERROR_EXTENDED_ERROR )
21.     {
22.         Win32::NetResource::WNetGetLastError( $Error, $Text,
$Provider );
23.         $Text .= " ($Provider)";
24.     }
25.     else
26.     {
27.         $Text = Win32::FormatMessage( $Error );
28.     }
29.     print "Error $Error: $Text";
30. }

```

[Example 2.16](#) shares a particular directory (`C:\TEMP`) using a name of `TempFiles` on `\server1`. If this is successful, you will be able to connect to `\server1\TempFiles`. If unsuccessful, a description of the error will be printed.

Keep in mind that creating a share whose name ends with a dollar sign (\$) will render the share invisible. When someone browses for a share to connect to, invisible shares are not displayed (although you can refer to my second book, *Win32 Perl Scripting: The Administrator's Handbook*, for a technique that displays all hidden shares on a machine). You can always, however, connect directly to an invisible share by specifying the UNC instead of browsing.

[Example 2.20](#) is similar to [Example 2.19](#) except that it uses the `Win32::Lanman` extension and specifies permissions that the share will have. `Win32::NetResource` is unable to specify such permission settings. Notice that line 6 references the `max_uses` key; this is different from the `maxusers` key that `Win32::NetResource` uses (`max_uses` is actually the more accurate because this is what the Win32 API uses).

Lines 13 through 19 use `Win32::Perms` to create a security descriptor. Line 19 assigns the security descriptor to the appropriate hash key, which is passed into the `NetShareAdd()` function in line 20. For more details on creating and managing security descriptors, see [Chapter 11](#).

### ***Example 2.20 Creating a shared directory with permissions***

```

01. use Win32::Lanman;
02. use Win32::Perms;
03. my %Share = (
04.     'path' => "C:\\Temp",
05.     'type' => Win32::Lanman::STYPE_DISKTREE,
06.     'max_uses' => -1,
07.     'netname' => "TempFiles",
08.     'remark' => "My Temporary Files",
09.     'passwd' => "",
10.     'permissions' => 0,
11. );
12. my $Machine = "\\\server1";

```

```

13. my $PermsObj = new Win32::Perms() || die;
14. $PermsObj->Allow( "Administrators", FULL_SHARE );
15. $PermsObj->Allow( "Domain Admins", FULL_SHARE );
16. $PermsObj->Allow( "Everyone", READ_SHARE );
17. $PermsObj->Deny( "Guest" );
18. $PermsObj->Deny( "Domain Guests" );
19. $Share{security_descriptor} = $PermsObj->GetSD( SD_RELATIVE );
20. if( Win32::Lanman::NetShareAdd( $Machine, \%Share ) )
21. {
22.     print "Successfully shared $Share{path} as
$Share{netname}.\n";
23. }
24. else
25. {
26.     print "Failed to share $Share{path} as $Share{netname}.\n";
27.     my $Error = Win32::Lanman::GetLastError();
28.     print "Error: ", Win32::FormatMessage( $Error );
29. }
30. $PermsObj->Close();

```

## Warning

*If you make a share invisible by providing a dollar sign (\$) at the end of its share name, it will be invisible to other Windows machines. It is important to know, however, that the name is not invisible to everyone. The invisibility feature is supported by Windows, but other applications and operating systems do not necessarily support it. Seagate's BackupExec backup program can discover all shares on a server, for example, as well as the UNIX SMB server called Samba.*

## Tip

*Not only can you hide a shared directory by appending a dollar sign (\$) to the end of its name, you can hide an entire computer from the network. By using the Win32::Lanman::NetServerSetInfo() function, you can set the hidden property to 0. This will prevent the machine from announcing its presence to the network, thus preventing users from seeing it when they browse the net. The machine still responds to queries, but typical browsing fails to display it. See the "[Configuring Servers](#)" section in [Chapter 3](#) for more details.*

## Note

*Currently, the Win32::NetResource extension is not capable of applying account-based permissions on a network share. Users who need to apply permissions can use the Explorer or Server Manager programs that come with Windows NT.*

## Removing Shared Network Resources

Now that you are familiar with creating shares, you will need to know how to remove or delete shares. This is not as complicated as creating shares. The function you use is `NetShareDel()`:

```
Win32::NetResource::NetShareDel( $ShareName [, $Machine] );
Win32::Lanman::NetShareDel( $Machine, $ShareName );
```

The `$ShareName` parameter is the name of the share. The `NetShareDel()` function is not case sensitive, so if you want to delete the share `\server1\TempFiles`, you can specify a `$ShareName` of `tempFILES` or `TempfileS` or any combination of case.

The `$Machine` parameter is the name of the machine that is to remove the share. The share must reside on the machine specified. The `Win32::Lanman` version of the function requires this parameter, but it is optional for the `Win32::NetResource` version. If it is not specified or it contains an empty string (""), the local machine is assumed.

If the `NetShareDel()` function is successful, it returns a 1; otherwise, it returns a 0. [Example 2.21](#) demonstrates how to remove a share.

### ***Example 2.21 Removing a shared resource from the network***

```
01. use Win32::NetResource;
02. if( Win32::NetResource::NetShareDel( "TempFiles",
"\\myserver" ) )
03. {
04.   print "Could not remove the share.\n";
05. }
```

## Checking on a Shared Network Resource

Suppose you need to check and make sure a particular device is being shared. You might want to check that your file server is sharing a particular directory or that your print server is sharing a particular printer, for example. You can do this using the `NetShareCheck()` function:

```
Win32::NetResource::NetShareCheck( $Device, $Type [, $Machine] );
Win32::Lanman::NetShareCheck( $Machine, $Device, \$Type );
```

This function checks to see whether `$Device` is being shared. A common example of `$Device` could be `D` or `C`. The device *must* be in capitals and can *only* be one character long. (Everything after the first character is ignored.)

### **Warning**

*I would strongly advise against using the `NetShareCheck()` functions.*

If the function is successful, the `$Type` parameter will be set to a value that the device claims to be. Notice that in the `Win32::Lanman` version, you have to pass in a reference to the `$Type` variable.

The `$Machine` parameter specifies which computer to check. If this value is an empty string or if it is not supplied, the local machine will be assumed.

Now that you know how the function is supposed to work, take a look at how it does work.

The `NetShareCheck()` function is an oddity. It makes use of the Win32 API function of the same name, which Microsoft has not documented well and *does not work as advertised*. If you try to check on a device that begins with a backslash, for example, the function will always claim that the device is a printer queue, even if the device does not exist! You can test this out for yourself by running the code in [Example 2.22](#).

### **Example 2.22 Checking whether a device is shared**

```
01. use Win32::Lanman;
02. my $Device = shift @ARGV || "C";
03. my %Values = (
04.     0 => "Disk", # STYPE_DISKTREE
05.     1 => "Printer", # STYPE_PRINTQ
06.     2 => "Communication Port", # STYPE_DEVICE
07.     3 => "IPC", # STYPE_IPC
08. );
09. my $Machine = "";
10. my $Type;
11. if( Win32::Lanman::NetShareCheck( $Machine, $Device, \$Type ) )
12. {
13.     print "The '$Device' device is a shared $$Values{$Type}.\n";
14. }
15. else
16. {
17.     print "The function failed.\n";
18. }
```

The `%Values` hash in [Example 2.22](#) contains values that are mentioned in neither the `Win32::Lanman` nor `Win32::NetResource` documentation. I got these values from the Win32 API documentation. These values are constants such as `STYPE_PRINTQ` and `STYPE_DISKTREE`, and they are not found in the `Win32::NetResource` extension for some reason (so I am using the values here rather than the constant names).

When you run the code, pass in a device (like a drive, directory, or printer device) as the first parameter. For example, the code

```
perl netcheck.pl C
```

on my machine will print

```
The 'C' device is a shared Disk..
```

because my `c:` drive is shared. (You must specify the drive letter as a capital character—lowercase `c` will fail.)

Because the function seems to recognize only the first character of the specified device, it is not clear how you can specify a printer, modem, or IPC device.

The only thing you can depend on when using this function is that if any directory is shared on a drive and you specify that drive as the device, the `Win32::NetResources::NetShareCheck()` function will succeed and the `$Type` parameter will be set to a disk type.

## Determining UNC's for Paths on Remote Drives

If you have a drive letter that has been mapped to a network sharepoint, you can discover what the UNC is for any path on that remote drive. This is done by using the `GetUNCName()` function:

```
Win32::NetResource::GetUNCName( $Unc, $Path );
```

The second parameter, `$Path`, is the path you want to translate into a UNC, such as `R:\Temp`. If the function is successful, the first parameter, `$Unc`, is set to the full UNC of `$Path`.

Assume, for example, that you have your `s:` drive mapped to `\server1\Graphics`, and you want to know what the full UNC would be to access the file `s:\Perl\Camel.gif`. You could try the code in [Example 2.23](#).

### ***Example 2.23 Retrieving the UNC name of a shared resource***

```
01. use Win32::NetResource;
02. my $Path = "S:\Perl\Camel.gif";
03. if( Win32::NetResource::GetUNCName( $Unc, $Path ) )
04. {
05. print "The UNC is $Unc\n";
06. }
```

The output for [Example 2.23](#) would look like this:

```
The UNC is \\server1\Graphics\PERL\CAMEL.GIF
```

So now you know that the next time you want to use `CAMEL.GIF`, you can tell your graphic editor to open `\server1\Graphics\PERL\CAMEL.GIF`. This is handy when a user moves from machine to machine (like a system administrator) and cannot depend on a particular network drive letter.

If the `GetUNCName()` function is successful, it returns a `true` value; otherwise, it returns `false`.

## Determining Shared Network Resources

Most people like to discover hidden treasures that lurk out there in cyberspace. Perl coders are no exception to this universal rule. This is one of the impetuses for the `GetSharedResources()`

function that will return a list of shared resources that exist on the network. The syntax for this function is as follows:

```
Win32::NetResource::GetSharedResources( \@List, $Type );
```

The first parameter (`\@List`) is a reference to an array. If the function is successful, this array is populated with `NETRESOURCE` hashes.

The second parameter (`$Type`) is the type of resource you are looking for. These types are defined under the `Type` key description of the `NETRESOURCE` hash.

If the `GetSharedResources()` function is successful, it returns a `1`; otherwise, it returns a `0`.

The `GetSharedResources()` function does not clean the array reference you pass into it (the first parameter). This means that if you call the function twice, your result will be the total results of both calls. This will most likely result in duplicate data; on a big domain, this can be very deadly to a Perl script because it will take up large amounts of memory.

## Tip

*It is best if you undefine the array reference before you call the `GetSharedResources()` function; otherwise, the results will be appended to the array. This could lead to a dramatically large array! For example, you would want to use code that is similar to this:*

```
undef @Resources;
Win32::NetResource::GetSharedResources( \@Resources,
RESOURCETYPE_DISK );
```

# Managing RAS Servers

For those of you who have RAS servers, it is awfully convenient to check on your modems to see who is logged on, which ports are in use, and who is allowed to dial in. These functions, among others, are available from the `Win32::RasAdmin` extension.

## Discovering Ports on the Server

When you have the urge to find out which ports on your RAS server are in use, you can sink your teeth into the `GetPorts()` function:

```
Win32::RasAdmin::GetPorts( $Server [ , \%Hash ] );
```

With this function, the `$Server` parameter is the proper name of the computer you want to query, as in `\server1`.

The optional second parameter (`\%Hash`) is a reference to a hash. If the function is successful, the hash reference will be filled out with a hash of hashes. The key names of the hash are the names of

the communication ports (such as `COM1`) available on the server. Each key has an associated value that is a `RASPORT` hash consisting of the values described in the list that follows:

- **DeviceType**. A description of the type of device connected to the port such as `Modem` or `ISDN`.
- **DeviceName**. The name of the device connected to the port, such as `Hayes 9600`.
- **MediaName**. The name of the media used for the connection to the port, such as `rassser` or `PCIMACISDN1`.
- **User**. The name of the client who has connected to the port.
- **Computer**. The name of the client's computer that has connected to the port.
- **Domain**. The domain that authenticated the user.
- **Flags**. A bitmask of values that describes the connection made to the port. Possible values are as follows:
  - **GATEWAY\_ACTIVE**. The server is using the NetBIOS gateway, allowing NetBIOS commands and messages to be passed to the client.
  - **MESSENGER\_PRESENT**. The client is running the NT Messenger service (so it can receive net messages).
  - **PORT\_MULTILINKED**. The port is multilinked with other ports, allowing multiple ports to act as one port (increasing bandwidth).
  - **PPP\_CLIENT**. The client is connected using the PPP protocol. If this is not set, the client connected using the SMB protocol.
  - **REMOTE\_LISTEN**. The NetBIOS gateway is set to RemoteListen.
  - **USER\_AUTHENTICATED**. The client connected to this port, and the user has been successfully authenticated.
- **StartTime**. The time when the client connected to the port. This value is the number of seconds since January 1, 1970. Because this is the same time format Perl uses, you can use the Perl time functions with this value.
- **Server**. If this flag is not zero, the server with which the port is associated is an NT Server; otherwise, it is an NT Workstation.
- **PortName**. The name of the port such as `COM1:`.

## Note

If the `GetPorts()` function is successful, it returns the number of ports found on the specified server; if the second parameter is supplied, the hash is populated with port information. If no ports were found (or if the server does not support RAS), it returns a 0.

## Note

If the `MESSENGER_PRESENT` and `REMOTE_LISTEN` flags are set, the client can receive network messages sent from the RAS server. If the `GATEWAY_ACTIVE` flag is also set, any machine can send messages to the client.

## Retrieving Port Information

When you are looking for comprehensive information about a particular port, such as what protocols are active on the port and the number of types transmitted, you use the `PortGetInfo()` function. Be prepared because this function retrieves quite a wealth of information:

```
Win32::RasAdmin::PortGetInfo( $Server, $Port, \%Hash );
```

The first parameter (`$Server`) is a proper RAS server name such as `\rasserver1`.

The second parameter (`$Port`) is the name of the port, such as `COM1:` or `COM32:`.

The third parameter (`\% Hash`) is a reference to a hash. If the function is successful, this hash is filled with a series of keys and subhashes. The same set of keys and values returned from the `GetPorts()` function is filled into this hash in addition to the following items:

- **LineCondition.** The condition or state of the port. The possible values are as follows:
  - **RAS\_PORT\_NON\_OPERATIONAL.** There is a problem on the port that is rendering the port nonoperational. The event log can be used to discover the reason for this failure.
  - **RAS\_PORT\_DISCONNECTED.** The port is not connected to any clients and is in a disconnected state.
  - **RAS\_PORT\_CALLING\_BACK.** The RAS server is currently calling back the client.
  - **RAS\_PORT\_LISTENING.** The port is waiting for a connection.
  - **RAS\_PORT\_AUTHENTICATING.** The server is currently authenticating the client.
  - **RAS\_PORT\_AUTHENTICATED.** The client has been successfully authenticated.
  - **RAS\_PORT\_INITIALIZING.** The device attached to the port (such as a modem) is currently being initialized. After the initialization has completed, the state will change to `RAS_PORT_LISTENING`.
- **HardwareCondition.** The state of the hardware attached to the port. This is usually a modem. The possible values are as follows:
  - **RAS\_MODEM\_OPERATIONAL.** The device is functioning correctly and is waiting for incoming calls.
  - **RAS\_MODEM\_HARDWARE\_FAILURE.** The device is not functioning correctly.
- **LineSpeed.** The speed at which the port and computer can communicate. This value is in bits per second.
- **Address.** This is a `RASPORTADDRESS` hash (refer to [Table 2.7](#)).
- **ConnectionStats.** A `RASPORTSTATS` hash (see [Table 2.8](#)).
- **PortStats.** A `RASPORTSTATS` hash (see [Table 2.8](#)). The hash pertains to the physical communication port, not the single connection on that port.
- **Parameters.** A hash that contains data that is pertinent to the particular media in use on the port.

The `PortGetInfo()` function will return a `1` if successful and a `0` if it failed.

**Table 2.7. Description of the `RASPORTADDRESS` Hash**

<b>Address Subhash Key</b>	<b>Description</b>
Nbf	This is the client's computer name. This key exists only if the client is successfully using NetBEUI.
Ip	This is the client's IP address. This key exists only if the client is successfully using TCP/IP.
Ipx	This is the client's IPX address. This key exists only if the client is successfully using IPX.

**Table 2.8. Description of the RASPORTSTATS Hash**

<b>ConnectionStats Subhash Key</b>	<b>Description</b>
BytesXmited	Total number of bytes transmitted from the port.
BytesRcved	Total number of bytes received by the port.
FramesXmited	Total number of frames transmitted from the port.
FramesRcved	Total number of frames received by the port.
CrcErr	Total number of CRC errors.
TimeoutErr	Total number of timeout errors.
AlignmentErr	Total number of alignment errors.
HardwareOverrunErr	Total number of hardware overruns. This indicates how many times the client has sent data too quickly for the server to read it.
FramingErr	Total number of framing errors.
BufferOverrunErr	Total number of times that the server's buffer was overrun (data coming in quicker than the server could handle).
BytesXmitedUncompressed	Total number of bytes transmitted that were uncompressed.
BytesRcvedUncompressed	Total number of bytes received that were uncompressed.
BytesRcvedCompressed	Total number of bytes received that were compressed.

## Retrieving Server Information

If you ever forget how a particular RAS server is configured, you can query it using the `ServerGetInfo()` function:

```
Win32::RasAdmin::ServerGetInfo( $Server, \%Info );
```

The first parameter (`Server`) is the proper name of a RAS server (something like `\rasserver1`).

The second parameter (`%Info`) is a hash reference. If the `ServerGetInfo()` function is successful, this hash is populated with the following keys:

- **Available.** The number of ports currently available for clients to connect to. This represents the number of ports configured for RAS that are not currently in use.
- **InUse.** The total number of ports currently connected with clients.
- **Version.** This is the version of RAS that the server is using. Possible values are as follows:
  - **RASDOWNLEVEL.** LAN Manager RAS server version 1.0
  - **RASADMIN\_35.** Windows NT 3.5 or 3.51 RAS server (or client)
  - **RASADMIN\_CURRENT.** Windows NT 4.0/2000 RAS server (or client) If the `ServerGetInfo()` function is successful, it returns `1`; otherwise, it returns `0`.

## Clearing Port Statistics

If you need to tell a port to reset its statistics, you can use the `ClearStats()` function:

```
Win32::RasAdmin::ClearStats( $Server, $Port );
```

The first parameter (`$Server`) is the proper name of the RAS server. The second parameter (`$Port`) is the port name such as `COM3:`.

If the `ClearStats()` function is successful, it returns `1` and all the port's statistics are reset; otherwise, it returns `0`.

## Disconnecting Clients

If a user has connected to your RAS server and you want to disconnect him (as a joke or if he has been wreaking terror and havoc on your network), you can force the port on which he is connected to disconnect using the `Disconnect()` function:

```
Win32::RasAdmin::Disconnect( $Server, $Port );
```

The first parameter (`$Server`), like the other `RasAdmin` functions, is the proper name of the RAS server. The second parameter (`$Port`) is the name of the port such as `COM27:`.

If the `Disconnect()` function is successful, it returns a `1` and disconnects the user; otherwise, it returns `0`.

## Case Study: Disconnecting a Particular User ID from All RAS Connections

Let's say your boss comes running up to you, as he does from time to time, and proclaims that someone has stolen his password. He fears that the hacker is dialing in and accessing files. He wants you to change his password, but you realize that changing the password will not do much if the hacker is already logged on (and authorization has been already granted). What you need to do is quickly go through every RAS server and disconnect any user who is logged on using your boss's user ID. Take a look at [Example 2.24](#); it demonstrates a simple way to accomplish this task.

### ***Example 2.24 Disconnecting a particular user ID from all RAS connections***

```

01. use Win32::NetAdmin;
02. use Win32::RasAdmin;
03. my $Userid = "Boss";
04. if( Win32::NetAdmin::GetServers( "", "", SV_TYPE_DIALIN,
05. \@List) )
05. {
06.     my @List;
07.     print "Searching each server for $Userid...\n";
08.     foreach my $Server ( @List )
09.     {
10.         my %Info;
11.         if( Win32::RasAdmin::GetPorts( $Server, \%Info ) )
12.         {
13.             foreach my $Port ( keys( %Info ) )
14.             {
15.                 if( $Info{$Port}->{User} =~ /^$Userid/i )
16.                 {
17.                     if( Win32::RasAdmin::Disconnect( $Server,
$Info{$Port}->{PortName} ) )
18.                     {
19.                         print "$Userid was disconnected from $Server port:
20. $Info{$Port}->{PortName}\n";
21.                     }
22.                 else
23.                 {
24.                     print "Could not disconnect $Userid from $Server
port:
25. $Info{$Port}->{PortName}\n";
26. # RasAdmin handles extended version of network
errors.
27.                     $Error = Win32::RasAdmin::GetErrorString(
28.                         Win32::RasAdmin::GetLastError() );
29.                     print "Error: $Error\n";
30.                 }
31.             }
32.         }
33.     }
34. }
35. }
36. else
37. {
38.     print "Could not find the RAS machine names.\n";
39.     print Win32::FormatMessage( Win32::NetAdmin::GetError() );
40. }

```

Here we are retrieving the list of RAS servers using `Win32::NetAdmin::GetServers()`. We then grab information on all the ports from each of the servers. If we find that a user is logged on to a port using the `$Userid` name, we disconnect him. Pretty cool, eh? Now, if you have hundreds of RAS servers located throughout the country, you can tell your boss that this little script will handle it and he can relax. There isn't even a need for overtime!

# Windows Terminal Server

The Windows Terminal Server (WTS) is a clever service that enables users to remotely connect to a machine and log on. For the most part, the user's experience is not much different from logging on to the machine directly from the keyboard. A remote user must run the Terminal Services Client application that, once connected to a WTS machine, displays the user's logon session.

You can think of WTS as being similar to an X-Windows server. It will send the commands to draw windows and icons to a remote display instead of a local display. This is what the client application does; it receives the window drawing commands that the local video card normally receives and renders them within the client application's window.

Because only the commands to draw the screen and keyboard and mouse inputs are sent over the network, the end experience is quite fast and reasonably responsive even over the slowest of network links.

## Tip

*When running the Terminal Services Client application, you can hit the CTRL+ALT+Break keys to expand the client application window to full screen. This better emulates the look and feel of the remote logon.*

The reason why WTS is inviting to administrators is that you can connect to any machine (that runs the Terminal Services service) and log on to administrate the box. This is seriously convenient when dealing with slow dialup connections. Terminal services (also known as *remote desktop* services in Windows XP) include the Microsoft Windows Terminal Services as well as Citrix's WinFrame services.

When you log on to a Terminal Server remotely, you see what appears to be a computer's desktop. This is a real desktop that is running on the remote machine (on the Terminal Server itself). This enables the user to select the Start button and select a program to run. Running such programs (for example, Microsoft Word) will cause the remote machine to run the program in its memory. The client sees the program running and can interact with it as if he were at the server's keyboard. You can think of this as if there are really long keyboard and mouse cords connected between the client's keyboard and mouse and the remote machine. Functionally, this is exactly what Terminal Services provide.

## Enumerating Terminal Servers

To locate all of the terminal servers in a particular domain, you can call either the `Win32::NetAdmin::GetServers()` or the `Win32::Lanman::NetServerEnum()` function (refer to the section "[Discovering Servers on the Network](#)" earlier in this chapter). They are both general-purpose functions that enable a script to locate different types of servers. However, there is a function that locates only WTS servers. This is the `WTSEnumerateServers()` function:

```
Win32::Lanman::WTSEnumerateServers( $Domain, \@List );
```

The first parameter (`$Domain`) is the name of the domain that you want to discover its terminal servers.

The second parameter (`\@List`) is a reference to an array. This array will be populated with a list of hash references. There is one hash reference for each WTS server discovered. Each hash reference contains only one key called `name`, which is the name of the WTS server.

If the function is successful, the specified array is populated with hash references representing each WTS server discovered, and the function returns `TRUE (1)`. Otherwise, the function returns `FALSE (0)`, and the array is cleared.

## Managing Sessions on a Terminal Server

When a Terminal Server client connects to a Terminal Server, it sees what appears to be a desktop. The fact of the matter is that it is indeed a real desktop. However, one desktop cannot see another desktop. That is, if one user clutters up his desktop with files, changes the screen saver, and modifies the background image, this does not affect the desktops of other users. This is because he is using his own *Windows Station*.

A Windows Station (also known as a WinStation) is an area of memory used to house a user's desktop. Every logged on user has a Windows Station assigned to him, regardless of whether he is using Terminal Services or not. When you log on using a machine's keyboard, it creates a Windows Station for you to connect to and interact with. Every service that runs on a machine has a Windows Station created for it (because all services must logon on with a user account).

Windows Stations define the environment with which a user will interact. This is where devices are attached such as the keyboard, mouse, and so on. When you log on to a Terminal Server, a Windows Station is created for you. The mouse and keyboard devices attached to the Windows Station are virtual and represent the remote mouse and keyboard from the network.

When you log on to your Windows Station, it loads your user's profile and creates a desktop based on your personalized settings. This desktop is contained within the Windows Station. All activity that occurs during a Terminal Server session does so within the Windows Station.

A Terminal Server session is really the existence of a Windows Station that someone has logged on to using Terminal Services. This also includes the Windows Station that a user might have logged on to through the server's keyboard.

### ***Listing Sessions in a Terminal Server***

You can obtain a list of all of the Terminal server sessions on a machine by calling the `WTSEnumerateSessions()` function:

```
Win32::Lanman::WTSEnumerateSessions( $Server, \@Sessions );
```

The first parameter (`$Server`) is the name of the Terminal server. If this is `undef` or an empty string, the function will use the current machine on which the script is running.

The second parameter (`\@Sessions`) is a reference to an array. This array will be populated with hash references. There is one hash reference per Terminal server session. Each hash reference will contain the keys listed in [Table 2.9](#).

If the function is successful in obtaining the list of sessions (even if the `@Sessions` array is empty—indicating that there are no sessions), it returns `TRUE (1)`. Otherwise, it returns `FALSE (0)`.

[Example 2.25](#) illustrates the use of this function in line 37.

**Table 2.9. Terminal Server Session Information**

Key Name	Description
<code>id</code>	A numeric value that uniquely identifies the session.
<code>state</code>	This is any one value from <a href="#">Table 2.10</a> .
<code>winstationname</code>	The name of the session's Window Station. This is basically a name used to describe the session.

**Table 2.10. Terminal Server Session States**

Constant Name	Description
<code>WTSActive</code>	A user has logged on to the Windows Station.
<code>WTSConnected</code>	A client is connected to the Windows Station.
<code>WTSConnectQuery</code>	The client is in the process of connecting to the Windows Station.
<code>WTSShadow</code>	The session is currently shadowing another Windows Station.
<code>WTSDisconnected</code>	The Windows Station is still active, but the client has disconnected.
<code>WTSIdle</code>	The existing Windows Station is waiting for a client to reconnect.
<code>WTSListen</code>	The Windows Station is waiting for a client to connect and log on.
<code>WTSReset</code>	The Windows Station is being reset.
<code>WTSDown</code>	The Windows Station has been taken down due to an error.
<code>WTSInit</code>	The Windows Station is initializing itself.

### Querying Session Information

You can query a particular session for particular information. This is done with the `WTSQuerySessionInformation()` function:

```
Win32::Lanman::WTSQuerySessionInformation( $Server, $Session,
\@Properties, \%Info );
```

The first parameter (`$Server`) is the name of the Terminal server. If this is `undef` or an empty string, the function will use the current machine on which the script is running.

The second parameter (`$Session`) is the session ID you want to query. Refer to the "[Listing Sessions in a Terminal Server](#)" section for details on enumerating the session IDs on a server.

The third parameter (`\@Properties`) is a reference to an array of properties that specify what information you are querying. This array can consist of any constants listed in [Table 2.11](#).

The fourth parameter (`\%Info`) is a reference to a hash. Each key in the hash represents one of the queried properties specified by the `\@List` array.

If the function is successful, it returns `TRUE (1)`; otherwise, it fails and returns `FALSE (0)`.

In [Example 2.25](#), the `WTSQuerySessionInformation()` function is used (line 46) to query all of the available information from each session on a particular server.

### ***Example 2.25 Displaying Terminal Server session information***

```
01. use Win32::Lanman;
02.
03. my %STATES = (
04.     &WTSActive => "User has logged on",
05.     &WTSConnected => "Connected",
06.     &WTSConnectQuery => "Connecting",
07.     &WTSShadow => "Shadowing",
08.     &WTSDisconnected => "Client disconnected",
09.     &WTSIdle => "Waiting for client to reconnect",
10.    &WTSListen => "Waiting for client to logon",
11.    &WTSReset => "Resetting",
12.    &WTSDown => "Down due to error",
13.    &WTSInit => "Initializing"
14.) ;
15.
16. my @QUERY = (
17.     WTSApplicationName,
18.     WTSClientAddress,
19.     WTSClientBuildNumber,
20.     WTSClientDirectory,
21.     WTSClientDisplay,
22.     WTSClientHardwareId,
23.     WTSClientName,
24.     WTSClientProductId,
25.     WTSConnectState,
26.     WTSDomainName,
27.     WTSInitialProgram,
28.     WTSOEMId,
29.     WTSSessionId,
30.     WTSUserName,
31.     WTSWinStationName,
32.     WTSWorkingDirectory
33. );
34.
35. my $Server = shift @ARGV || "";
36. my @SessionList;
```

```

37. Win32::Lanman::WTSEnumerateSessions( $Server, \@SessionList )
|| die ;
38. foreach my $Session ( @SessionList )
39. {
40.     my %Info;
41.     print "\n", "-" x 30, "\n";
42.     print "Session: $Session->{id}\n";
43.     print "State: $STATES{$Session->{state}}\n";
44.     print "WinStation: $Session->{winstationname}\n";
45.     next if( $Session->{state} == WTSListen );
46.     if( Win32::Lanman::WTSQuerySessionInformation( $Server,
$Session->{id}, \@QUERY,
    \%Info ) )
47.     {
48.         my( $Net, $IPStruct ) = unpack( "La*", $Info{clientaddress} );
49.         my( $IP ) = join( ".", (unpack( "C*", $IPStruct ))[2..5] );
50.         my( $DisplayX,
51.             $DisplayY,
52.             $DisplayColors ) = unpack( "L3",
$Info{clientdisplay} );
53.         $DisplayColors <= 2;
54.
55.         print "\tClient IP: $IP\n";
56.         print "\tClient Display: $DisplayX x $DisplayY
($DisplayColors bits of color)\n";
57.         foreach my $Key ( sort( keys( %Info ) ) )
58.         {
59.             print "\t$key: $Info{$Key}\n";
60.         }
61.     }
62. }
63.
64. sub Error()
65. {
66.     print Win32::FormatMessage( Win32::Lanman::GetLastError() );
67. }

```

**Table 2.11. Terminal Server Query Options**

Constant Name	Description
WTSApplicationName	The name of the application that the session is currently running. If the script runs in the Terminal Services console, this value is empty.
WTSClientAddress	The network address of the client. This is a packed structure that can be unpacked using:

```

my( $Net, $IPStruct ) = unpack( "La*", $Info{clientaddress} );
my( $IP ) = join( ".", (unpack( "C*", $IPStruct ))[2..5] );

```

**Table 2.11. Terminal Server Query Options**

Constant Name	Description
	<pre>) )[ 2 .. 5 ] ) if( 2 == \$Net );</pre>
WTSClientBuildNumber	The build number of the Terminal Services client application running on the client machine. This does not indicate the version number, only the build number.
WTSClientDirectory	The directory from which the Terminal Services client application runs. This represents the directory on the remote client (not server) that the Terminal Services client program (usually <code>MSTSC.EXE</code> ) resides on.
WTSClientDisplay	The size of the client's display. This describes the client machine's screen capabilities. This is a packed structure that can be unpacked using:
	<pre>( \$X, \$Y, \$Colors ) = unpack( "L3", \$Info{clientdisplay} ); \$Colors &lt;= 2;</pre>
	<p><code>\$X</code> and <code>\$Y</code> represent the display's resolution, while <code>\$Colors</code> represents how many color bits there are per pixel.</p>
WTSClientHardwareId	An ID number that represents the Terminal Services client application's hardware. The meaning of this value is unknown.
WTSClientName	The name of the client machine connected to the Terminal Server.
WTSClientProductId	An ID number that represents the Terminal Services client's product identifier. The meaning of this value is unknown.
WTSConnectState	The current state of the client's connection. This is any one value from <a href="#">Table 2.10</a> .
WTSDomainName	The name of the domain on which the logged-on user resides.
WTSInitialProgram	The path to an application that the logged-on user is configured to automatically run when logging on.
WTSOEMId	A string that represents any OEM identifier.
WTSSessionId	The session identifier.
WTSUserName	The name of the logged-on user.
WTSWinStationName	The name of the session's Windows Station.
WTSWorkingDirectory	The default directory used when launching the program specified by the <code>WTSInitialProgram</code> property.

### ***Sending Messages to Users in a Session***

Once you know which session a particular user logged in on, you can send her a message that is displayed in her session. This is done with the `WTSSendMessage()` function:

```
Win32::Lanman::WTSSendMessage( $Server, $Session, $Title, $Message  
[, $Style [,  
$Timeout [, \$Response] ] ] );
```

The first parameter (`$Server`) is the name of the terminal server. If this is `undef` or an empty string, the function will use the current machine on which the script is running.

The second parameter (`$Session`) is the session ID you want to query. Refer to the "[Listing Sessions in a Terminal Server](#)" section for details on enumerating the session IDs on a server.

The third parameter (`$Title`) is the title that will be displayed in the message box.

The fourth parameter (`$Message`) is the full text message to be displayed in the message box.

The fifth parameter (`$Style`) is optional. This determines how the message box will look and function. This is identical to the `$Flags` parameter of the `Win32::MsgBox()` function (see the section "[Displaying Message Boxes](#)" in [Chapter 9, "Console"](#)). This value is the result of OR'ing a message box button flag (see [Table 9.17](#)), an icon flag (see [Table 9.18](#)), and a modality flag (see [Table 9.19](#)). If this parameter is not specified, the `MB_OK` flag is used.

The sixth parameter (`$Timeout`) indicates how long (in seconds) the function is to wait for the session's user to respond to the dialog box. If this value is `0`, the function returns immediately, and there is no way to determine the user's response to the dialog box. If the timeout value has been exceeded, the function returns setting the `$Response` parameter with `IDTIMEOUT`. If this parameter is not specified, the function returns immediately and no user response is recorded.

The seventh parameter (`\$Response`) is optional. This parameter is a reference to a scalar variable. The user's response to the dialog box is stored in this variable. This value can be either `IDTIMEOUT` (indicating that the user waited too long to respond) or any value from [Table 9.20](#).

If the function is successful, it returns `TRUE (1)`, even if the timeout value has been exceeded. Otherwise, it is `FALSE (0)`.

In [Example 2.26](#), a message is sent to all users logged in to a Terminal Server. In this particular example, each user will suddenly see a dialog box asking whether he or she would like a latte drink. The script waits 10 seconds for the user to respond. After that time, the dialog disappears and the `WTSSendMessage()` function returns. The various message box flag values (lines 4 through 7) were discussed in [Chapter 9](#).

### **Example 2.26 Sending messages to Terminal Server clients**

```
01. use Win32::Lanman;  
02. my $Server = lc shift @ARGV || "";  
03. my @SessionList;  
04. my $MB_YESNO = 0x04;  
05. my $MB_ICONQUESTION = 0x20;  
06. my $IDYES = 0x06;  
07. my $IDNO = 0x07;
```

```

08. Win32::Lanman::WTSEnumerateSessions( $Server, \@SessionList )
|| die Error();
09. foreach my $Session ( @SessionList )
10. {
11.     my %Info;
12.     my $Response;
13.     next if( WTSListen == $Session->{state} );
14.     next if( "console" eq lc $$Session->{winstationname} );
15.     Win32::Lanman::WTSQuerySessionInformation( $Server,
16.                                                 $Session->{id},
17.                                                 [WTSUserName],
18.                                                 \%Info );
19.     print "Asking \u$Info{username} on server: $Server...\n";
20.     Win32::Lanman::WTSSendMessage( $Server,
21.                                     $Session->{id},
22.                                     "Coffee break",
23.                                     "Would you like a latte?",
24.                                     $MB_YESNO | $MB_ICONQUESTION,
25.                                     10,
26.                                     \$Response );
27.     if( IDTIMEOUT == $Response )
28.     {
29.         print "\t\u$Info{username} waited too long, he forfeits his
latte.\n";
30.     }
31.     elsif( $IDYES == $Response )
32.     {
33.         print "\tYes. \u$Info{username} wants a latte.\n";
34.     }
35.     elsif( $IDNO == $Response )
36.     {
37.         print "\tNo. \u$Info{username} does not want a latte.\n";
38.     }
39.     print "\n";
40. }
41.
42.sub Error
43.{
44.
return( Win32::FormatMessage( Win32::Lanman::GetLastError() ) );
45.}

```

## ***Disconnecting Terminal Server Sessions***

Terminal Services enables a logged-on server to disconnect from the session. This means that the user's session and Windows Station remain intact, but the network connection to the session is disconnected. This functionality enables users to start an application that requires considerable time to process and then disconnect. The application continues to process on the remote Terminal Server machine, even though the client has disconnected (possibly even shutting down his client machine). When the user connects again, he continues the previous session.

A script can force a user to disconnect by calling the `WTSDisconnectSession()` function:

```
Win32::Lanman::WTSDisconnectSession( $Server, $Session [,  
$bWait] );
```

The first parameter (`$Server`) is the name of the Terminal Server where the session exists. If this is `undef` or an empty string, the local machine is used.

The second parameter (`$Session`) is the identifier of the session you want to disconnect. Refer to the section "Listing Sessions in a Terminal Server" for details.

The optional third parameter (`$bWait`) is a Boolean flag that indicates whether or not the function returns immediately. If this parameter is `true`, the function will wait until the session has disconnected before returning. If this parameter is `false`, the function returns immediately, possibly before the session has been disconnected from the client. If this parameter is not specified, a `false` value is used.

The function returns a `true` (1) if successful; otherwise, it returns `false` (0).

The user account running the script that calls this function requires the Disconnect permission.

[Example 2.27](#) shows how to use this function. Line 6 collects the list of all terminal servers in the domain. Then, for each terminal server, line 10 obtains a list of all of its sessions. Line 14 skips a session if it is already in a disconnected state; this prevents the script from wasting time querying for information on the state.

Line 15 queries the session for the logged-in username. If the name matches the specified username (both converted to lowercase to make it easier to match), line 23 actually disconnects the client from the session. Notice that the `WTSDisconnectSession()` function passes in a `false` value for the `$bWait` parameter. This causes the function to return immediately, not waiting for the client to actually disconnect. This is quite useful if you have many servers with many sessions to check. Otherwise, the process could take much longer than it needs to.

### ***Example 2.27 Disconnecting a user from all Terminal Servers***

```
01. use Win32::Lanman;  
02. my $User = lc shift @ARGV || "";  
03. my @SessionList;  
04. my @ServerList;  
05. my $Domain = Win32::DomainName();  
06. Win32::Lanman::WTSEnumerateServers( $Domain, \@ServerList );  
07. foreach $Server ( @ServerList )  
08. {  
09.     print "Querying $Server->{name}\n";  
10.    Win32::Lanman::WTSEnumerateSessions( $Server->{name},  
11.                                         \@SessionList ) || next;  
12.    foreach my $Session ( @SessionList )  
13.    {  
14.        my %Info;  
15.        next if( WTSDisconnected == $Session->{state} );
```

```

15.      if( Win32::Lanman::WTSQuerySessionInformation( $Server-
>{name} ,
16.                                         $Session-
>{id} ,
17.                                         [WTSUserName] ,
18.                                         \%Info ) )
19.      {
20.          if( $User eq lc $Info{username} )
21.          {
22.              print "Disconnecting $Info{username} from session
$Session->{id} on $Server-
>{name}.\n";
23.          Win32::Lanman::WTSDisconnectSession( $Server->{name} ,
$Session->{id} , 0 );
24.          }
25.      }
26.  }
27. }

```

### **Forcing Terminal Server Sessions to log off**

Just as a script can disconnect a client from a terminal session, a script can force a logged-on user to log off. This is performed by calling the `WTSLogoffSession()` function:

```
Win32::Lanman::WTSLogoffSession( $Server, $Session [, $bWait] );
```

The first parameter (`$Server`) is the name of the Terminal Server where the session exists. If this is `undef` or an empty string, the local machine is used.

The second parameter (`$Session`) is the identifier of the session you want to disconnect. Refer to the section "[Listing Sessions in a Terminal Server](#)" for details.

The optional third parameter (`$bWait`) is a Boolean flag that indicates whether or not the function returns immediately. If this parameter is `true`, the function will wait until the session has fully logged off before returning. If this parameter is `false`, the function returns immediately, possibly before the session has logged off. If this parameter is not specified, a `false` value is used.

The function returns a `true (1)` if successful; otherwise, it returns `false (0)`.

The user account running the script that calls this function requires the Reset permission if it is logging off clients other than itself. The function cannot log off users who are logged on interactively through the console (keyboard) on the server locally.

You can verify that the session was successfully logged off by calling the `WTSQuerySessionInformation()` function. Pass in the ID of the session you are logging off. The function returns `false (0)` if the session no longer exists (has been logged off).

Example 2.28 accepts the name of a Terminal Server (line 3). It then logs off all users who have connected from remote machines (line 16). Notice that lines 8 and 9 skip over all sessions that are either listening for new clients to connect or logged in via the console.

### ***Example 2.28 Logging off all remote Terminal Server clients***

```
01. use Win32::Lanman;
02. my @SessionList;
03. my $Server = shift @ARGV || "";
04. Win32::Lanman::WTSEnumerateSessions( $Server, \@SessionList )
|| die Error();
05. foreach my $Session ( @SessionList )
06. {
07.     my %Info;
08.     next if( WTSListen == $Session->{state} );
09.     next if( "console" eq lc $$Session->{winstationname} );
10.     if( Win32::Lanman::WTSQuerySessionInformation( $Server,
11.                                                     $Session-
>{id},
12.                                                     [WTSUserName],
13.                                                     \%Info ) )
14.     {
15.         print "Logging off $Info{username} from session $Session-
>{id} on $Server.\n";
16.         Win32::Lanman::WTSLogoffSession( $Server, $Session->{id},
0 );
17.     }
18. }
19.
20. sub Error()
21. {
22.     print Win32::FormatMessage( Win32::Lanman::GetLastError() );
23. }
```

## **Managing Processes on a Terminal Server**

Because users logged on to a Terminal Server can execute various programs, it might be important for an administrator to manage these different applications. For one reason or another, a process might freeze or start to consume too much memory. These types of problems might require a network administrator to root out the problem and correct it before the server itself is impacted.

### ***Listing Processes on a Terminal Server***

To discover the comprehensive list of all processes running on a Terminal Server, you can use the `WTSEnumerateProcesses()` function:

```
Win32::Lanman::WTSEnumerateProcesses( $Server, \@Processes );
```

The first parameter (`$Server`) is the name of the Terminal Server where the processes exists. If this is `undef` or an empty string, the local machine is used.

The second parameter (`\@Processes`) is a reference to an array. This array will be populated with hash references. There will be one hash per process. Each hash will contain the following keys:

- **sessionid**. The session identifier under which the processes runs.
- **processid**. The identifier of the process.
- **name**. The name of the process.
- **sid**. The security identifier of the account under which the process is running. See [Chapter 11](#) for more details on SIDs.

If the function is successful, it returns `TRUE (1)` and populates the array. Otherwise, it returns `FALSE (0)`.

See [Example 2.29](#) for an example of how this function is used.

### ***Terminating a Process on a Terminal Server***

If you know the process identifier of a particular process, a script can terminate the process using the `WTSTerminateProcess()` function:

```
Win32::Lanman::WTSTerminateProcess( $Server, $Pid, $ExitCode );
```

The first parameter (`$Server`) is the name of the Terminal Server where the processes exists. If this is `undef` or an empty string, the local machine is used.

The second parameter (`$Pid`) is the process's unique identifier. This can be found by calling `WTSEnumerateProcesses()`.

The third parameter (`$ExitCode`) specifies the exit code that the process will return when it is terminated. All processes can return a numeric value when they terminate. This is the value that the process will report.

If the function is successful, it returns `TRUE (1)` and the process is terminated. Otherwise, the function returns `FALSE (0)`.

[Example 2.29](#) illustrates how to call both the `WTSEnumerateProcesses()` and `WTSTerminateProcess()` functions. This example will terminate all instances of a specified program running on all Terminal Services in a domain.

### ***Example 2.29 Terminating all instances of an application***

[View full width]

```
01. use Win32::Lanman;
02. my $Application = lc shift @ARGV || die;
03. my @ServerList;
04. my $Domain = Win32::DomainName();
05. Win32::Lanman::WTSEnumerateServers( $Domain, \@ServerList );
06. foreach $Server ( @ServerList )
07. {
08.     my @ProcessList;
09.     print "Querying $Server->{name}\n";
```

```

10.    Win32::Lanman::WTSEnumerateProcesses( $Server->{name} ,
11.        \@ProcessList ) || next ;
12.    {
13.        my $User;
14.        my $Terminated = "Ignored";
15.        if( 0 == $Process->{processid} )
16.        {
17.            $Process->{name} = "Idle thread";
18.        }
19.        if( "" ne $$Process->{sid} )
20.        {
21.            my @UserList;
22.            if( Win32::Lanman::LsaLookupSids( $Server->{name} ,
23.                [$Process->{sid}] ,
24.                \@UserList
25.            ) )
26.            {
27.                $User = "$UserList[0]->{domain}\\" if( "" ne
28.                    $$UserList[0]->{domain} );
29.                $User .= $UserList[0]->{name};
30.            }
31.            if( $Application eq lc $Process->{name} )
32.            {
33.                if( Win32::Lanman::WTSTerminateProcess( $Server->{name} ,
34.                    $Process->{processid} ,
35.                    0 ) )
36.                {
37.                    $Terminated = "Terminated";
38.                }
39.                printf( " %%19s %05d %- 30s %s %s\n" , $Terminated ,
40.                    $Process->{processid} , $User ,
41.                    $Process->{name} );
}

```

## Configuring User Accounts to Use Terminal Services

Each user can be configured to have specific Terminal Server options. This is done with the `WTSQueryUserConfig()` and `WTSSetUserConfig()` functions:

```

Win32::Lanman::WTSQueryUserConfig( $Server , $User , \@Properties ,
    \%Config );
Win32::Lanman::WTSSetUserConfig( $Server , $User , \%Config )

```

The `$Server` parameter is the name of either a domain or a Terminal Server. If this is `undef` or an empty string, the function will use the current machine on which the script is running.

The `$User` parameter is the name of the user account to be used.

The `\@Properties` parameter is a reference to an array. Populate this array with any of the values in [Table 2.12](#).

The `\%Config` parameter is a reference to a hash. Populate this hash with the values you want to modify. Each hash key identifies the property value by name. The name is determined by removing the `WTSUserConfig` from its related setting constant name (refer to [Table 2.12](#)) and then converting the remaining string to lowercase. For example, the key for the `WTSUserConfigInitialProgram` setting would be `initialprogram`. The key for the `WTSUserConfigfAllowLogonTerminalServer` setting would be `fallowlogonterminalserver`.

If these functions are successful, they return `TRUE (1)`; otherwise, they return `FALSE (0)`.

**Table 2.12. Terminal Server Settings for User Accounts**

Constant Name	Description
<code>WTSUserConfigInitialProgram</code>	The path to an application that will run when the user logs on to a Terminal Server. The path is relative to the Terminal Server onto which the user logs.
<code>WTSUserConfigWorkingDirectory</code>	This is the default directory used when running the application specified in the <code>WTSUserConfigInitialProgram</code> property.
<code>WTSUserConfigfInheritInitialProgram</code>	This is a Boolean value that indicates whether the client can specify the initial program ( <code>WTSUserConfigInitialProgram</code> ).  A <code>0</code> value indicates that the user cannot specify the initial program. If an initial program is configured, <i>only</i> that program can run. When the program terminates, the user is forced to log off.  A <code>1</code> value indicates that the user can specify the initial program.
<code>WTSUserConfigfAllowLogonTerminalServer</code>	This is a Boolean value that indicates whether the user is allowed to log on to a Terminal Server.
<code>WTSUserConfigTimeoutSettingsConnections</code>	This is the maximum time (in milliseconds) that the user can remain logged on to a Terminal Server. The user is alerted to the pending timeout one minute before it occurs.

**Table 2.12. Terminal Server Settings for User Accounts**

Constant Name	Description
<code>WTSUserConfigTimeoutSettingsDisconnects</code>	When time expires, the user is either disconnected or terminated, depending on the <code>WTSUserConfigBrokenTimeoutSettings</code> setting. Every time the user logs on, the timer is reset.
<code>WTSUserConfigTimeoutSettingsIdle</code>	A value of <code>0</code> means that there is no timeout imposed.
<code>WTSUserConfigfDeviceClientDrives</code>	The amount of time (in milliseconds) that a disconnected session is allowed to exist before it is terminated. A value of <code>0</code> means that a disconnected session can remain indefinitely.
<code>WTSUserConfigfDeviceClientPrinters</code>	The amount of time (in milliseconds) that a session is allowed be idle (no keyboard or mouse input). When this time has been exceeded, the session is either disconnected or terminated, depending on the value of the <code>WTSUserConfigBrokenTimeoutSettings</code> setting.
<code>WTSUserConfigfDeviceClientDefaultPrinter</code>	A value of <code>0</code> means that a session can remain idle indefinitely.
<code>WTSUserConfigBrokenTimeoutSettings</code>	A Boolean value indicating whether the terminal server automatically reestablishes client drive mappings at logon. <i>This applies only to Citrix clients.</i>
<code>WTSUserConfigReconnectSettings</code>	A Boolean value indicating whether the terminal server automatically reestablishes client printer mappings at logon. <i>This applies only to Citrix clients.</i>
	A Boolean value indicating what happens when the connection or idle timers expire or when a connection is lost due to a connection error. A value of <code>0</code> will disconnect sessions. A value of <code>1</code> will terminate sessions.
	A Boolean value indicating how disconnected clients can reconnect. A value of <code>0</code> means a client can reconnect to a session from any machine. A value of <code>1</code> means a client can only reconnect to a session from the remote machine

**Table 2.12. Terminal Server Settings for User Accounts**

Constant Name	Description
WTSUserConfigModemCallbackSettings	with which she originally created the session. Attempting to reconnect from a different machine will result in a new logon session.
WTSUserConfigModemCallbackPhoneNumber	How a Terminal Server reacts to a dialup connection. <i>This applies only to Citrix clients.</i>
WTSUserConfigShadowingSettings	A value of <b>0</b> means callbacks are disabled. A value of <b>1</b> means the user is prompted for a phone number to call back. Any value in the <b>WTSUserConfigModemCallbackPhone_Number</b> property will displayed as the default phone number. A value of <b>2</b> means the call is disconnected, and the server calls the user back using the phone number specified in the <b>WTSUserConfigModemCallbackPhone_Number</b> property.
WTSUserConfigTerminalServerProfilePath	Phone number used for dialup connection callbacks. Refer to the <b>WTSUserConfigModemCallbackSettings</b> property. <i>This applies only to Citrix clients.</i>
WTSUserConfigTerminalServerHomeDir	Determines whether or not the user can be shadowed. Shadowing enables a user to monitor another user's activity. <i>This applies only to Citrix clients.</i>
WTSUserConfigTerminalServerHomeDirDrive	A value of <b>0</b> means shadowing is disabled. A value of <b>1</b> means the shadow is allowed input, and the user is notified of the shadow. A value of <b>2</b> means the shadow is allowed to provide input, and the user is not notified of the shadow. A value of <b>3</b> means the shadow is not allowed to provide input, and the user is notified of the shadow. A value of <b>4</b> means the shadow is not allowed to provide input, and the user is not notified of the shadow.
Create PDF with GO2PDF for free, if you wish to remove this line, click here to buy Virtual PDF Printer	The path to the user's profile that is used during a Terminal Server session. This profile must be created manually and exist before the logon process.
Create PDF with GO2PDF for free, if you wish to remove this line, click here to buy Virtual PDF Printer	The path to the user's home directory that is used during a Terminal Server session. This can be either a local path or a UNC.
Create PDF with GO2PDF for free, if you wish to remove this line, click here to buy Virtual PDF Printer	A drive letter to which a UNC specified in the <b>WTSUserConfigTerminalServerHomeDir</b>

**Table 2.12. Terminal Server Settings for User Accounts**

Constant Name	Description
<code>WTSUserConfigTerminalServerRemoteHomeDir</code>	property is mapped.
<code>WTSUserConfigTerminalServerHomeDir</code>	A Boolean value indicating whether or not the path specified in the <code>WTSUserConfigTerminalServerHomeDir</code> property is a UNC path (which will be mapped to a drive letter) or a local directory.  A value of <code>0</code> means the path is a local directory. A value of <code>1</code> means the path is a UNC and needs to be mapped to the drive letter specified by the <code>WTSUserConfigTerminalServerHomeDirDrive</code> property.

## Waiting for Terminal Server Events

A script can monitor a Terminal Server for particular events. These events indicate that something occurred on the Terminal Server, enabling the script to respond to such actions. This is done by calling the `WTSWaitSystemEvent()` function:

```
Win32::Lanman::WTSWaitSystemEvent( $Server, $Mask, \$Flags );
```

The first parameter (`$Server`) is the name of the Terminal Server where the processes exists. If this is `undef` or an empty string, the local machine is used.

The second parameter (`$Mask`) is a bitmask representing the various events being monitored. This can be any combination of values listed in [Table 2.13](#), OR'ed together.

The third parameter (`\$Flags`) is a reference to a scalar variable. This is a bitmask just like `$Mask`, except that this parameter represents the events that occurred. To discover what events occurred, you can AND this parameter with the values found in [Table 2.13](#).

This function will not return until at least one of the monitored events occurs. There is no timeout value for this function.

If the function is successful, it returns `true` (1) and waits for the specified events. Otherwise, it returns `false` (0).

[Example 2.30](#) demonstrates how a script can monitor a Terminal Server. Line 5 waits for a client to either connect or disconnect. When this occurs, the script prints out which event took place. The script loops repeating this forever.

### **Example 2.30 Monitoring client connections**

```
01. use Win32::Lanman;
02. my $Server = lc shift @ARGV || die;
```

```

03. my $Result;
04. print "Monitoring Terminal Server events for server
$Server...\n";
05. while( Win32::Lanman::WTSWaitSystemEvent( $Server,
06.                                         WTS_EVENT_DISCONNECTWTS_EVENT_CONNECT,
07.                                         \$Result ) )
08. {
09.   if( $Result & WTS_EVENT_CONNECT )
10.  {
11.    print "\tA Terminal Server client connected to $Server.\n";
12.  }
13.  if( $Result & WTS_EVENT_DISCONNECT )
14.  {
15.    print "\tA Terminal Server client disconnected from
$Server.\n";
16.  }
17. }

```

**Table 2.13. Terminal Server Settings for User Accounts**

<b>Constant</b>	<b>Description</b>
WTS_EVENT_ALL	Wait for any event type.
WTS_EVENT_CONNECT	A client connected to a Windows Station.
WTS_EVENT_CREATE	A new Windows Station was created.
WTS_EVENT_DELETE	An existing Windows Station was deleted.
WTS_EVENT_DISCONNECT	A client disconnected from a Windows Station.
WTS_EVENT_LICENSE	The Terminal Services' license state changed. This occurs when a license is added or deleted using License Manager.
WTS_EVENT_LOGOFF	A user logged off from either the Terminal Services console or a client Windows Station.
WTS_EVENT_LOGON	A user logged on to the system from either the Terminal Services console or a client Windows Station.
WTS_EVENT_RENAME	An existing Windows Station was renamed.
WTS_EVENT_STATECHANGE	A Windows Station's connection state changed. This is any value from <a href="#">Table 2.10</a> .

## Summary: Perl and Win32 Networks

With all the hype surrounding networking on the Win32 platform, it is easy to forget that quite a bit of management needs to be handled. Win32 Perl is there for just such purposes.

The basic necessities to manage a network of computers are available using the common Win32 extensions. From RAS server management to network resource sharing, all these things are possible.

It is possible to use non-Perl utilities to accomplish these same tasks. As someone explained to me via email, however, you cannot guarantee that each computer on which you run your Perl script will have each and every one of the utilities you need. Likewise, it is difficult to depend on the utilities because occasionally a utility will change the way it works from version to version. By using a Perl extension, you can always depend on the extension to exist within your Perl library tree.

Perl can be quite a workhorse that can administer your network, saving you quite a bit of money in professional administrative utilities.

## Chapter 3. Administration of Machines

Administrating a network of machines is quite a large task that can be difficult no matter how the operating system attempts to simplify its interface. Several disciplines must be mastered to keep the network free from problems. Simple concepts such as user accounts, machine accounts, trusts, shares, permissions, and privileges become quite tedious to manage when you are contending with hundreds or thousands of users and machines.

This is really not so much of an issue for the user who is running Windows ME at home. This user probably will never deal with a network other than the Internet via a dialup modem. For this scenario, much of this chapter really will not be of use other than for those with a curious nature.

This chapter addresses the Win32 Perl extensions that administrators can use to manage machines. This ranges from creating user accounts and groups (so that users can log on to a machine) to RAS permissions. Configuring a machine by means of INI files and the Registry is also discussed, as is looking for events in the Event Log.

With the advent of Active Directory, Perl scripts need to rely on using the `Win32::OLE` extension and ADSI to manage AD-based user and group accounts. Refer to [Chapter 5](#), "Automation," for details on using `Win32::OLE`.

### User Account Management

One of the most important elements in a domain is the user account. Without these accounts, no users could log on to the domain, and without users logging on, no work would get done. Even if your domain is totally automated and needs no user intervention, you still need some accounts. You always need an Administrator account, for example, so you can log on and manage the domain. Likewise, many services need accounts to log on to the machine or domain to perform some tasks.

#### Testing for User Account Existence

Before creating a new user account, you should ensure that a user account with the same name you intend to create does not already exist. This can be done using `UsersExist()` function:

```
Win32::NetAdmin::UsersExist( $Machine, $User );
```

The first parameter (`$Machine`) is the name of the machine to be checked for the account. The machine name must be a proper machine name (prepended with double backslashes). If you are

checking on a domain, you will want to specify a domain controller (primary or backup). If this is an empty string, the local machine is assumed.

The second parameter (`$User`) is the name of the account to be checked.

If the account does exist, the function returns a `true` value; otherwise, it returns `false`. Examples [3.1](#) and [3.4](#) show the `UsersExist()` function in action.

## Creating User Accounts

Creating user accounts is rather straightforward; you use the `Win32::NetAdmin`'s `UserCreate()` function, the syntax for which is as follows:

```
Win32::NetAdmin::UserCreate( $Machine, $User, $Password,
$PasswordAge,
$Privileges, $HomeDir, $Comment, $Flags, $ScriptPath );
```

If the syntax for `UserCreate()` appears to be overwhelming, don't fret; it's much simpler than you might think. The nine parameters that you need to pass into this function are described in detail in the text that follows.

The first parameter (`$Machine`) is the server. This is just the name of the machine in which this new user account will reside. If you are creating an account in a domain, you need to specify the primary (or backup) domain controller. If you are adding the user to your `Accounting` domain, for example, you could use the `Win32::NetAdmin::GetDomainController()` function to find the primary domain controller for the domain. You would then use that machine as the server for this function. If this parameter is left blank (""), the account is created on the local machine. If you have the appropriate permissions, you could specify a computer name such as `\\\Workstation`, which would attempt to create the user account on that particular machine.

The second parameter (`$User`) is the new user ID. This is what the user would enter when logging on. This user ID is not case sensitive and can be up to 256 characters long.

### Tip

*For those who are concerned with network security when passing passwords over a network, it is good to know that passwords are encrypted when performing a logon. These passwords, however, are fairly easy to crack if you have a network protocol analyzer. The techniques to crack NT passwords are documented on several hacker pages on the World Wide Web.*

*If you truly want to secure passwords from even network sniffers, you should make sure that all passwords are more than seven characters in length. Passwords with eight or more characters encrypt differently than those of up to seven characters. This is so that passwords are backward compatible with Microsoft's LAN Manager.*

The third parameter for `UserCreate()` (`$Password`) is the new password for the user account. Note that after this account has been created, you cannot use this function to change the password. The length of the password cannot be more than 14 characters.

## Note

The 14-character limit is defined by the Win32 API and could change in future versions of Windows. If you are a C programmer, this limit is defined in the `LMCONS.H` header file.

The fourth parameter for `UserCreate()` (`$PasswordAge`) is supposed to represent the number of seconds since the password was last changed. This parameter is ignored, and there have been questions as to why it is even a part of this function. It is possible that some future version of NT might use it, but as of this writing, there is no need for it. This parameter can be any value, but it is probably best to keep it as `0` or an empty string.

## Note

The Win32 API requires the `$PasswordAge` parameter when creating a new user account; however, it is not used, and the password age is managed automatically by the operating system. This parameter might be included in the function for backward-compatibility issues. Whoever ported the function to Perl most likely did not consider the fact that it was not necessary to include it.

The fifth parameter for `UserCreate()` (`$Privileges`) represents privileges that a user might need. A user account can have a few different privileges, but when creating an account, you must use the `USER_PRIV_USER` privilege. [Table 3.1](#) documents the full list of privileges.

**Table 3.1. User Privileges to Be Used with the `Win32::NetAdmin:: CreateUser()` Function**

Privilege	Assignment
<code>USER_PRIV_GUEST</code>	Assigned to guest accounts
<code>USER_PRIV_USER</code>	Assigned to regular user accounts
<code>USER_PRIV_ADMIN</code>	Assigned to administrator accounts

A user account's privilege can be changed by group management (such as adding the user account to the Administrators group). The privileges are hierarchical, so if the `JOEL` account is in the Domain Admins, Domain Users, and Domain Guests groups, the privilege that `JOEL` will hold is `USER_PRIV_ADMIN`.

The sixth parameter for `UserCreate()` (`$HomeDir`) is the user's home directory. All NT accounts have a home directory and can be either a local directory such as `d:\users\joel` or a UNC such as `\server2\home\users\joel`. Not everyone makes use of this, but it can be quite handy for users who need to have a location on a file server where they can put their files. This is especially important when a company has several PCs and a user can log on to any one of them, making it a necessity to access personal files from anywhere. (This is sometimes known as *hoteling*, in which a user uses a different workstation every day, as is popular in data processing centers.)

The seventh parameter for `UserCreate() ($Comment)` serves as a comment. This comment does not have any practical use other than associating a string with the user account. Usually administrators make use of this to make sense of an account. An account called `IUSR_MACHINE` could have a comment of "`The Web Server Account`", for example, which makes more sense to a human than the user ID does. This string is practically unlimited in size and can be an empty string ("").

The eighth parameter for `UserCreate() ($Flags )` specifies special flags for the account. These flags consist of account options (such as those shown by the User Manager when displaying a user's properties) and account types. This parameter can be any combination of the options listed in [Table 3.2](#) and any one from [Table 3.3](#). These values are all logically OR'ed together.

**Table 3.2. User Account Flags**

<b>Flag</b>	<b>Description</b>
<code>UF_ACCOUNTDISABLE</code>	Disables the account. When used with the <code>Win32::NetAdmin::UserCreate()</code> function, the account will be created but disabled.
<code>UF_DONT_EXPIRE_PASSWD</code>	The password for the account never expires.
<code>UF_HOMEDIR_REQUIRED</code>	A home directory is required. (This flag is ignored in Windows NT.)
<code>UF_PASSWD_CANT_CHANGE</code>	The user cannot change the password; only administrators and account operators can change it.
<code>UF_PASSWD_NOTREQD</code>	No password is required on this account. This overrides any policy that requires all accounts to have passwords and passwords that are a particular size.
<b>Flag</b>	<b>Description</b>
<code>UF_LOCKOUT</code>	The account is locked out. This is caused by too many incorrect attempts to log on. This flag cannot be set, but it can be cleared if already set.
<code>UF_SCRIPT</code>	This indicates that the logon script was executed. Usually you would check for this flag on an account if you wanted to know whether the account's logon script was executed the last time  the user logged on. For some reason, the Win32 API requires that this flag must be set when creating an account.

**Table 3.3. Account Types Specified by a User and Computer Account's Flags**

Account Type	<b>Description</b>
<code>UF_INTERDOMAIN_TRUST_ACCOUNT</code>	Used between the domains that indicate a <i>domain trust</i> . This account is used by a primary domain controller. A domain that trusts another domain will have an account of this type, and the other domain's PDC will log on using it.
<code>UF_NORMAL_ACCOUNT</code>	Used for a <i>global user account</i> . This is the type of user account

**Table 3.3. Account Types Specified by a User and Computer Account's Flags**

Account Type	Description
UF_TEMP_DUPLICATE_ACCOUNT	that is typically created.
UF_SERVER_TRUST_ACCOUNT	Indicates a <i>local user account</i> . The user can use this account to access the primary domain but no domains that trust this domain.
UF_WORKSTATION_TRUST_ACCOUNT	Backup domain controllers have a computer account of this type. This type of account indicates that this machine is a BDC for the primary domain.
UF_SCRIPT	This is the type of account that a server (not a domain controller) or a workstation computer has. If a computer logs on to a domain using an account of this type, it is a member of the domain. This is used only with computer accounts.

### Note

*When creating a user account, you must specify an account type (typically UF\_NORMAL\_ACCOUNT) and at least the UF\_SCRIPT option. These two are the minimum flags that must be used; otherwise, the UserCreate() function will fail. You need to logically OR these values together, such as:*

```
$Flags = UF_SCRIPT | UF_NORMAL_ACCOUNT;
```

The ninth parameter for `UserCreate()` (`LogonScript`) is the logon script. This is the path to a program or batch file that will be executed when the user is logging on with this account. Any valid local path or UNC is allowed, as is an empty string ("").

### Note

*Only administrators and account operators can successfully call the Win32::NetAdmin::UserCreate() function.*

### Tip

*Creating a user account could fail if the password does not comply with password policies imposed by the server or the domain.*

The code in [Example 3.1](#) illustrates the use of the `UserCreate()` function.

### **Example 3.1 Using the `UserCreate()` function**

```

01. use Win32::NetAdmin;
02. use Win32::AdminMisc;
03. my $User = "Joel";
04. my $FullName = "Joel Smith";
05. my $Domain = Win32::DomainName();
06. my $Server = "";
07. my %Account = (
08.     password => "MySecurePass",
09.     homedir => "\\\HomeServer\Home\$User",
10.    priv => USER_PRIV_USER,
11.    flags => UF_SCRIPT | UF_NORMAL_ACCOUNT,
12.    fullname => $FullName,
13.    comment => "The account for $FullName",
14.    logon => "Logon.pl"
15. );
16. Win32::NetAdmin::GetDomainController( '', $Domain, $Server );
17. if( ! Win32::NetAdmin::UsersExist( $Server, $User ) )
18. {
19.     if( Win32::NetAdmin::UserCreate( $Server,
20.                                     $User,
21.                                     $Account{password},
22.                                     0,
23.                                     $Account{priv},
24.                                     $Account{homedir},
25.                                     $Account{comment},
26.                                     $Account{flags},
27.                                     $Account{logon} ) )
28.     {
29.         my $Result =
Win32::AdminMisc::UserSetMiscAttributes( $Server,
30.                                         $User,
USER_FULL_NAME=>$Account{fullname} );
31.         if( $Result )
32.         {
33.             print "Successfully added the $User account.\n";
34.         }
35.         else
36.         {
37.             print "Failed to successfully add the $User account.\n";
38.         }
39.     }
40. }
41. else
42. {
43.     print "The account already exists.\n";
44. }

```

## Creating Machine Accounts

It might be interesting to know that when an NT machine is registered as part of a domain, a machine account is created just as a user account is created for a new user. This machine account is

not much different from a user account. In fact, the Win32 API uses the same function to create machine accounts that it uses to create user accounts. This means that you can create a machine account by calling `Win32::NetAdmin::UserCreate()`. To do this, you must specify the `UF_WORKSTATION_ACCOUNT` flag from [Table 3.3](#). This will work not only for workstations but for adding backup domain controllers to a domain as well as adding interdomain trusts by specifying the corresponding account type flag `UF_INTERDOMAIN_TRUST_ACCOUNT`, listed in [Table 3.3](#).

Unlike creating a normal user account, you must follow some tricks to create a machine account. The following list describes these tricks:

- The first parameter *must* point to the primary domain controller.
- The account type used in the eighth parameter must be `UF_WORKSTATION_TRUST_ACCOUNT` (or whatever appropriate value you require for the type of account you are creating).
- The name (the second parameter) must be the computer's name in all uppercase characters, and the last character must be a dollar sign (\$). The computer name `\BETTYS-COMPUTER` would use the name `BETTYS-COMPUTER$`.
- The name must not be more than 15 characters long. (This does not include the trailing dollar sign.)
- The password (the third parameter) must be the name of the computer in all lowercase characters (as opposed to the account name) and must not include a trailing dollar sign. This password cannot be more than 14 characters. If the computer name contains more than 14 characters, truncate the password at the fourteenth character. To use the preceding example, the password for `\BETTYS-COMPUTER` would be "bettys-compute".
- The user calling a script that creates machine accounts needs to have the Administrator privilege on the target computer specified in the first parameter.

After a workstation's (or server's, or controller's, or domain trust's) account has been created, you need to log the machine on to the domain. The first time the new machine is logged on, it negotiates with the PDC for a new password that will be used every time the new machine logs on. This happens automatically when a machine logs on to a domain in which a new account has been created. The PDC detects that the machine is logging on for the first time and initiates the changes.

After a machine account has been created (either manually by a Perl script or by using any domain administration tool), the account can be managed as if it were a regular user account. The account can be renamed, the password can be changed (be very careful with this because you cannot tell a machine which password to use—only set it to the original password setting as described in the preceding list), it can be deleted, and the accounts can be listed using Perl functions found in `Win32::NetAdmin` and `Win32::AdminMisc`. If you manage machine accounts, you must follow the rules outlined in the preceding list.

As a side note, only administrators and those users who have been granted the `SeMachineAccountPrivilege` (also known as "Add Workstations to Domain" in the User Manager) can create and manage machine accounts.

## Changing User Account Configuration

After an account has been created, it can be manipulated. The various properties that make up an account can be both queried and set by using the `Win32::NetAdmin` extension's `UserGetAttributes` and `UserSetAttributes` functions:

```

Win32::NetAdmin::UserGetAttributes($Machine, $UserName, $Password,
$PasswordAge,
$Privilege, $HomeDir, $Comment, $Flags, $ScriptPath);
Win32::NetAdmin::UserSetAttributes( $Machine, $UserName, $Password,
$PasswordAge,
$Privilege, $HomeDir, $Comment, $Flags, $ScriptPath);

```

The first parameter (`$Machine`) is the machine from which you want to retrieve the account information. This should be your primary domain controller, but any domain controller will do. (If you use a backup domain controller, it will just take longer for your other DCs to learn about any updates to the account.) If this string is empty, the local machine is assumed. This string should be formatted as a proper machine name with double backslashes, as in `\PrimaryDC`.

The second parameter (`$UserName`) is the name of the user account.

The third parameter (`$Password`) is the password for the account. This is only used with the `UserSetAttributes()` function. The Win32 API does not support retrieving passwords.

The fourth parameter (`$PasswordAge`) is the age of the password. This indicates how many seconds have passed since the last time the password was changed. This value is automatically set by NT and cannot be set, only retrieved.

The fifth parameter (`$Privilege`) is the privilege that the account has been granted. This can be any one value from [Table 3.1](#).

The sixth parameter (`$HomeDir`) is the home directory for the account. This is any full path or UNC. It can also be an empty string.

The seventh parameter (`$Comment`) is the comment field for the account. This can be an empty string.

The eighth parameter (`$Flags`) represents the account's option flags. These flags can consist of any number of flags specified in [Table 3.2](#) and one flag from [Table 3.3](#). If you are setting this, these options are logically OR'ed together. If you are testing for these flags, you need to logically AND this parameter with the constant to test whether the flag is set. (A `TRUE` result indicates that the flag is set.)

The ninth parameter (`$ScriptPath`) is the full path (either local path or UNC) of the account's logon script. This can be an empty string.

If the functions are successful, they return a `TRUE` value; otherwise, they return `FALSE`.

`UserSetAttributes()` will update the account with the information passed into the function.

`UserGetAttributes()` will retrieve the values from the account that the passed-in parameters represent, as in [Example 3.2](#).

### ***Example 3.2 Using Win32::NetAdmin::UserGetAttributes()***

```

01. use Win32::NetAdmin;
02. use Win32::AdminMisc;
03. my $User = "Joel";
04. my $Domain = Win32::DomainName();

```

```

05. my $Server = "";
06. my( $Password, $PassAge, $Privilege,
07.      $HomeDir, $Comment, $Flags, $ScriptPath );
08. Win32::NetAdmin::GetDomainController( "", $Domain, $Server );
09. if( Win32::NetAdmin::UserGetAttributes( $Server, $User,
$Password,
10.     $PassAge, $Privilege, $HomeDir, $Comment, $Flags,
$ScriptPath ) )
11. {
12.   print "The user '$User' has a home directory of
'$HomeDir'.\n";
13. }

```

The `Win32::AdminMisc` extension has identical functions, with the exception that there is an added parameter. The third parameter is the account's full name. This is generally the full name of the user whom the account represents.

These functions and their kin in the `Win32::AdminMisc` extension are pretty much made obsolete by two functions in the `Win32::AdminMisc` extension: `UserGetMiscAttributes()` and `UserSetMiscAttributes()`. These two functions retrieve and set several different options, ranging from the necessary and obvious to the obscure and vague:

```

Win32::AdminMisc::UserGetMiscAttributes( $Server, $User,
\%Attribs );
Win32::AdminMisc::UserSetMiscAttributes( $Server, $User,
$Option=>$Attribute[ ,
$Option2=>$Attribute2[ , ... ] ] );

```

Both of these functions specify a first parameter (`$Server`) that is the name of the machine or the domain from which to retrieve the information. If this is an empty string (""), the domain currently logged on to is assumed. If a machine is specified, you need to prefix the machine's name with double forward- or backslashes, as in `\ServerA` or `//ServerB`. All the functions in the `Win32::AdminMisc` extension allow for this capability to use either forward- or backslashes. Be aware, however, that this is not a practice found in most other extensions.

The second parameter (`$User`) is the name of the user account.

For `UserGetMiscAttributes()`, the third parameter (`\%Attribs`) is the last one passed into the function. This parameter is a reference to a hash. If successful, this hash will be populated with the keys described in [Table 3.4](#).

For `UserSetMiscAttributes()`, all parameters starting with `$Option=> $Attribute` must be specified in pairs. These pairs are formatted as either:

`attribute, value`

or

```
attribute=>value
```

The format used is not important (because both of them are equivalent), but the second is preferred because it is easier to see what attribute is associated with which value.

When setting attributes, you need only to specify the attributes you are changing—unlike the `NetAdmin` and `AdminMisc`'s `UserSetAttributes()`, in which you need to specify all values.

Both functions return a `true` value if successful and a `false` value if they fail.

**Table 3.4. User Account Attributes Used with `Win32::AdminMisc::UserGetMiscAttributes()` and `Win32::AdminMisc::UserSetMiscAttributes()`**

Attribute	Description
<code>USER_ACCT_EXPIRES</code>	Specifies when the account will expire. This value is stored as a number of seconds elapsed since 00:00:00 January 1, 1970. A value of <code>TIMEQ_FOREVER</code> indicates that the account never expires. Note that this is the same time format that Perl uses, so you can pass this value into Perl's time functions, such as <code>localtime()</code> . To set the time to expire in one week, you could use <code>time() + (7 * 24 * 60 * 60)</code> .
<code>USER_AUTH_FLAGS</code>	The user's operator privileges. This is a read-only value and cannot be changed using the <code>Win32::AdminMisc</code> 's <code>SetUserMiscAttributes()</code> function. This value is based on membership in local groups. If the user is a member of the Print Operations group, <code>AF_OP_PRINT</code> is set. If the user is a member of the Server Operations group, <code>AF_OP_SERVER</code> is set.  If the user is a member of the Account Operations group, <code>AF_OP_ACCOUNTS</code> is set. The value of <code>AF_OP_COMM</code> is never set. (It looks like it was designed for future versions of NT.)
<code>USER_BAD_PW_COUNT</code>	Specifies the number of times the user tried to log on to this account using an incorrect password. A value of <code>0xFFFFFFFF</code> indicates that the value is unknown for some reason. This attribute is readonly and is maintained separately on each domain controller in the domain. To get an accurate value, each domain controller in the domain must be queried, and the sum is used.
<code>USER_CODE_PAGE</code>	The code page for the user's language of choice. Windows NT/2000/XP does not use this code page, but it is included for backward compatibility.
<code>USER_COMMENT</code>	The user account comment.
<code>USER_COUNTRY_CODE</code>	The country code for the user's language of choice. Windows NT does not use this country code, but it is included for backward compatibility.
<code>USER_FLAGS</code>	This value consists of several flags that determine features of the user account. <a href="#">Table 3.2</a> lists these flags. This parameter also describes the type of account being described. <a href="#">Table 3.3</a> lists a description of account types.

**Table 3.4. User Account Attributes Used with Win32::AdminMisc::UserGetMiscAttributes() and Win32::AdminMisc::UserSetMiscAttributes()**

<b>Attribute</b>	<b>Description</b>
USER_FULL_NAME	The full name of the user.
USER_HOME_DIR	The path of the home directory for the user specified.
USER_HOME_DIR_DRIVE	This specifies the drive letter assigned to the user's home directory. This is used primarily for logon purposes.
USER_LAST_LOGOFF	Specifies when the last logoff occurred. This value is stored as the number of seconds elapsed since 00:00:00, January 1, 1970. A value of zero means that the last logoff time is unknown. This attribute is read-only. This attribute is maintained separately on each domain controller in the domain. To get an accurate value, each domain controller in the domain must be queried, and the largest value is used.
USER_LAST_LOGON	Specifies when the last logon occurred. This value is stored as the number of seconds elapsed since 00:00:00, January 1, 1970. This attribute is readonly. This attribute is maintained separately on each domain controller in the domain. To get an accurate value, each domain controller in the domain must be queried, and the largest value is used.
Attribute	Description
USER_LOGON_HOURS	Points to a 21-byte (168-bit) bit string that specifies the times during which the user can log on. Each bit represents a unique hour in the week. The first bit (bit 0, word 0) is Sunday, 0:00 to 0:59; the second bit (bit 1, word 0) is Sunday, 1:00 to 1:59; and so on.
USER_LOGON_SERVER	The name of the machine to which logon requests are sent. Machine names are preceded by two backslashes (\ \ ). When a machine name is represented by an asterisk (\ \ *), the logon request can be handled by any logon server. An empty string indicates that requests are sent to the domain controller. This attribute is read-only. For Windows NT Servers, <code>UserGetMiscAttributes()</code> will return \ \ * for global accounts.
USER_MAX_STORAGE	Specifies the maximum amount of disk space the user can use. Use the value specified by <code>USER_MAXSTORAGE_UNLIMITED</code> to use all available disk space.
USER_NAME	Specifies the name of the user account. This is a read-only value and cannot be set by the <code>UserSetMiscAttributes()</code> function. If you want to change the username, you need to use the <code>Win32::AdminMisc::Rename()</code> function.
USER_NUM_LOGONS	Counts the number of successful times that the user tried to log on to this account. A value of <code>0xFFFFFFFF</code> indicates that the value is unknown. This attribute is read-only and is maintained separately on each domain controller in the domain. To get an accurate value, each domain controller in the domain must be queried, and the sum of all the values is used.

**Table 3.4. User Account Attributes Used with Win32::AdminMisc::UserGetMiscAttributes() and Win32::AdminMisc::UserSetMiscAttributes()**

Attribute	Description
USER_PARMS	This is set aside for use by applications. This string can be an empty string, or it can have any number of characters. Microsoft products use this attribute to store user configuration information. Altering this value is not recommended.
USER_PASSWORD	Specifies a one-way, encrypted, LAN Manager 2.x-compatible password. This is not retrieved due to the Win32 API. It is here for backward compatibility with LAN Manager. For all practical purposes, this attribute is ignored and can be neither set using the <code>UserSetMiscAttributes()</code> function nor retrieved with the <code>UserGetMiscAttributes()</code> function.
USER_PASSWORD_AGE	Specifies the number of seconds elapsed since the password was last changed. This value is managed automatically by NT and cannot be set.
USER_PASSWORD_EXPIRED	Determines whether the password of the user has expired. For <code>UserGetMiscAttributes()</code> , this attribute will be zero if the password has not expired (will be and nonzero if it has). For <code>UserSetMiscAttributes()</code> , specify nonzero to indicate that the user must change his password at next logon and specify zero to turn off the message indicating that the user must change password at next log on. If this parameter is set to a nonzero value, the user will only be able to log on interactively, at which time he will be forced to change passwords. Logging on as a process (noninteractively) will fail.
USER_PRIMARY_GROUP_ID	Specifies the relative ID (RID) of the Primary Global Group for this user. This attribute must be <code>DOMAIN_GROUP_RID_USERS</code> (defined in <code>NTSEAPI.H</code> for you C coders). This attribute must be the RID of a global group in which the user is enrolled. For most Perl scripts, this value is ignored.
USER_PRIV	Consists of one of three values that specify the level of privilege assigned to the user account. Refer to <a href="#">Table 3.1</a> for information on the privileges.
USER_PROFILE	A path to the user's profile. This can be an empty string, a full local path, or a UNC.
USER_SCRIPT_PATH	The path of the user's logon script, <code>.COM</code> , <code>.CMD</code> , <code>.EXE</code> , or <code>.BAT</code> file. Basically any executable program can be placed here. If your machine has mapped the <code>.PL</code> extension to execute <code>PERL.EXE</code> , this value could specify a Perl script (such as <code>LOGON.PL</code> ). This value can be an empty string to specify that there is no logon script.
USER_UNITS_PER_WEEK	Specifies the number of equal-length time units into which the week is divided. This attribute uses these time units to compute the length of the bit string in the <code>USER_LOGON_HOURS</code> attribute. This value must be <code>UNITS_PER_WEEK</code> for LAN Manager 2.0. This attribute is read-only. For Windows NT services, the units must be one of the following:

**Table 3.4. User Account Attributes Used with Win32::AdminMisc::UserGetMiscAttributes() and Win32::AdminMisc::UserSetMiscAttributes()**

Attribute	Description
	SAM_DAYS_PER_WEEK, SAM_HOURS_PER_WEEK, or SAM_MINUTES_PER_WEEK.
USER_WORKSTATIONS	This is a list of workstation and server names from which the user can log on. Up to eight machine names can be listed, and they must be separated by commas (,). An empty string/value indicates that there is no restriction. To disable logons from all workstations to this account, set the UF_ACCOUNTDISABLE value in USER_FLAGS attribute.
USER_USER_ID	This is a relative ID (RID) that identifies the user account within the domain in which it resides. This is a read-only value.
USER_USR_COMMENT	This is an arbitrary text string used as a comment. This is commonly not displayed and is used for internal comments about the particular user account.

In [Example 3.3](#), the `UserGetMiscAttributes()` and `UserSetMiscAttributes()` functions are used to enable any account in the "Accounting" global group that has been previously disabled. Notice that the `USER_FLAGS` attribute is logically AND'ed with the `UF_ACCOUNTDISABLE` constant to determine whether the account has been disabled. To turn off the account disabled bit, you need to be careful not to alter any other bits. This is done by AND'ing the `USER_FLAGS` value with the *two's complement* of the `UF_ACCOUNTDISABLE` constant. If you are not familiar with two's complement, consider it to be the binary inverse of a number. If I specify ~4 (the two's complement of four), for example, I am referring to the binary value "`11111011`", which is the inverse of the binary value for 4: "`00000100`".

By using the two's complement, you can be sure to turn off only the `UF_ACCOUNTDISABLE` bit; this is the same as enabling the account.

### **Example 3.3 Using AdminMisc's `UserGetMiscAttributes()` and `UserSetMiscAttributes()`**

```

01. use Win32::NetAdmin;
02. use Win32::AdminMisc;
03. my $Group = Win32::DomainName();
04. my @Members;
05. if( Win32::NetAdmin::GroupGetMembers( "", $Group, \@Members ) )
06. {
07.     foreach my $User ( @Members )
08.     {
09.         my %Attrs;
10.         Win32::AdminMisc::UserGetMiscAttributes( "", $User,
11. \%Attrs );
12.         # Check to see if the account disabled bit is set
13.         if( $Attrs{USER_FLAGS} & UF_ACCOUNTDISABLE )
14.         {

```

```

15.      # The account is disabled so re-enable it by ANDing the
16.      # flags attribute
17.      # with the two's complement of the disable account flag.
This will
18.      # turn off only the UF_ACCOUNTDISABLE bit but leave
other bits as
19.      # they are
20.      $Flags = $Attrs{USER_FLAGS} & ~UF_ACCOUNTDISABLE;
21.      Win32::AdminMisc::UserSetMiscAttributes( "",           ,
22.                                              $User,
23.
USER_FLAGS=>$Flags );
24.    }
25.  }
26. }

```

## Note

*To set the `USER_PASSWORD_EXPIRED` flag, the account's password must be able to expire. See the `UF_DONT_EXPIRE_PASSWD` value under the `USER_FLAGS` attribute for more details. You cannot specify zero to negate the expiration of a password that has already expired.*

## Renaming User Accounts

Quite often, there is a need to rename an account. If a user changes his or her name for any reason (such as a marriage), for example, there might be a need to rename the user ID. You could just create a new user account, but that would mean having to reconfigure it, copy over the old profile, add the account to any needed groups, reapply permissions onto directories and files... oh, just so much work! It is much easier to just rename the account.

By renaming the account, all configuration information remains the same, as do all file permissions and group memberships. In fact, renaming an account only changes the user ID name; everything else is left as it originally was.

The `RenameUser()` function facilitates this need:

```
Win32::AdminMisc::RenameUser( $Machine, $User, $NewUser );
```

The first parameter (`$Machine`) is the name of the machine or domain where the user account resides. If this is an empty string, the domain currently logged on to is assumed.

The second parameter (`$User`) is the name of the user account that will be renamed.

The third parameter (`$NewUser`) is the new name of the user ID.

The use of this function requires Administrator or Account Operator privileges. If successful, the function returns a TRUE value; otherwise, it returns FALSE . [Example 3.4](#) shows an example of how to use the `RenameUser()` function.

#### **Example 3.4 Renaming a user account**

```
01. use Win32::NetAdmin;
02. use Win32::AdminMisc;
03. my $User = "Joel";
04. my $NewUser = "Jane";
05. my $Domain = Win32::DomainName();
06. my $Server = "";
07. Win32::NetAdmin::GetDomainController( '', $Domain, $Server );
08. if( Win32::NetAdmin::UsersExist( $Server, $User ) )
09. {
10.   if( Win32::AdminMisc::RenameUser( $Server, $User,
$NewUser ) )
11.   {
12.     print "The account '$User' was renamed to '$NewUser'.\n";
13.   }
14. }
15. else
16. {
17.   print "Could not rename account '$User' because it does not
exist.\n";
18. }
```

## **Managing User Passwords**

Passwords are always a problem for any administrator and user. The user must be forced to remember a countless number of passwords, some of which he has no control over. The administrator is always resetting passwords for users who have forgotten them. There are a few ways to manage this hassle with the Win32 extensions.

The first of the password functions (which checks whether a password is correct or not) can be run by anyone (no special privileges are required):

```
Win32::AdminMisc::UserCheckPassword( ($Machine | $Domain), $User,
$Password );
```

The first parameter (`$Machine` or `$Domain`) is the domain name or proper machine name where the user account resides. If a domain name is specified, the domain's PDC will be looked up and used. If this is an empty string, the domain currently logged on to is assumed.

The second parameter (`$User`) is the name of the user account. If this is an empty string, the account that is running the script is assumed.

The third parameter (`$Password`) is the password for the user account.

If the password for the specified user account is correct, the function returns a `TRUE` value; otherwise, `FALSE` is returned. It is important to know that this function will check the password by attempting to change the password to a new password (which is the same password).

`UserCheckPassword()` does this just as if it were calling

`Win32::AdminMisc::UserChangePassword()` but passing in the same value for both the old and new passwords. This can be a problem on some systems if they do not allow reuse of passwords or if the account is not allowed to change passwords.

Several functions actually provide the capability to change the password for an account. The first two are identical to each other:

```
Win32::NetAdmin::UserChangePassword( ($Machine | $Domain), $User,
$OldPassword,
$NewPassword );
Win32::AdminMisc::UserChangePassword( ($Machine | $Domain), $User,
$OldPassword,
$NewPassword );
```

Both functions are identical and either can be used.

In both of these functions, the first parameter (`$Machine` or `$Domain`) is the domain name or proper machine name where the user account resides. If this is an empty string, the domain currently logged on to is assumed.

The second parameter (`$User`) is the name of the user account. If this is an empty string, the account that is running the script is assumed.

The third parameter (`$OldPassword`) is the current password for the user account.

The fourth parameter (`$NewPassword`) is the new password.

If the password is successfully changed, both of these functions return `TRUE`; otherwise, they return `FALSE`.

The new password is limited to the policies placed on user accounts such as limits to the number of characters used, whether old passwords can be reused, and whether the account is allowed to change passwords. Anyone can change any account password with these functions, provided he knows the current password for the account.

Administrators usually have to reset passwords for users who have forgotten theirs. This can be done by using the `SetPassword()` function:

```
Win32::AdminMisc::SetPassword( ($Machine | $Domain), $User,
$NewPassword );
```

The first parameter (`$Machine` or `$Domain`) is the proper machine name or the domain where the account resides. An empty string indicates the domain currently logged on to.

The second parameter (`$User`) is the name of the user account whose password will be changed.

The third parameter (`$NewPassword`) is the new password.

The `SetPassword()` function will work only if the user calling it has administrative rights on the machine or domain that houses the user account. It will return `TRUE` if the password was successfully changed; otherwise, it returns `FALSE`. Just like the other password functions, policy restrictions can cause this function to fail.

## Removing User Accounts

When an account is no longer needed, it can be removed from the user account database. It is usually a better practice to disable the account rather than delete it. This is because if, for whatever reason, you need the account later, you can reactivate it and its configuration as it was when disabled. When an account is disabled, no one can log on using that account; for all practical purposes, it might as well be deleted. For more details on deleting accounts and the consequences of doing so, see the following Note.

### Note

*Disabling an account is an alternative to deleting one. When an account is deleted, literally all information related to the account is lost. This includes file ownership and permissions, user privileges, group memberships, Registry permissions, and share accesses, to name a few.*

*Suppose your manager quits one day and your company hires another one. If you deleted the preceding manager's account, you would have to re-create a new one that would have to be totally reconfigured. If you just disabled the preceding manager's account, he could not log on or access his old data. When the new hire starts, you could just rename the disabled account, change the password, and reactivate the account. This way, the new manager has her new account with the exact same privileges, permissions, and file ownerships that the preceding manager had.*

*Disabling an account can be accomplished by setting the `UF_ACCOUNTDISABLED` bit on the `USER_FLAGS` attribute. Refer to `Win32::AdminMisc::UserSetMiscAttributes()` for more details.*

If the account no longer will be used, it can be deleted with the `UserDelete()` function:

```
Win32::NetAdmin::UserDelete( ($Machine | $Domain), $Account );
```

The first parameter (`$Machine`) is the proper name of the machine from which to remove the account. If the specified machine is a domain controller, the account will be removed from the domain that the server represents. If the machine specified is a workstation or a server, the account is deleted from the machine's local account database. If an empty string ("") is passed in, the account is removed from the current domain to which the machine is logged on.

The second parameter (`$Account`) is the name of the account to be removed.

If the account is successfully deleted, the function returns a `TRUE` value; otherwise, `FALSE` is returned.

### Tip

*When removing a user account from a domain, it is best to point the first parameter to the primary domain controller for the domain. If the account is removed from a backup domain controller (BDC), the BDC must be synchronized with the primary domain controller (PDC) for the account to truly be removed. Then it must be synchronized from the PDC to the remaining BDCs in the domain. This can take some time unless you force domain resynchronization using the Server Manager program. This means that even though you delete an account, someone might log on using it 10 minutes later because he logged on to a domain controller that had not been updated.*

*When an empty string is specified as the first parameter to the `Win32::NetAdmin::UserDelete()` function, an attempt will be made to delete the account on the PDC.*

## Managing User RAS Attributes

When an administrator needs to manage user accounts and whether they have access to RAS and any of its attributes, he can use the `Win32::RasAdmin` extension's `UserGetInfo()` function:

```
Win32::RasAdmin::UserGetInfo( ($Machine | $Domain), $User,  
\%Info );
```

The first parameter (`$Machine` or `$Domain`) is the proper machine or domain name from which you want to get the list of groups. If you specify a domain name, it will use either the PDC or a BDC (whichever is most readily available) for the specified domain. If this is an empty string, a domain controller for the domain currently logged on to will be used.

The second parameter (`$User`) is the name of the user account.

The third parameter (`\%Info`) is a reference to a hash that will be emptied and then populated with RAS attributes.

If `UserGetInfo()` is successful, the hash is populated with values found in [Table 3.5](#) and returns `TRUE`; otherwise, it returns `FALSE`. If the function fails, the hash is emptied.

The user's RAS attributes can be changed using the `UserSetInfo()` function:

```
Win32::RasAdmin::UserSetInfo( ($Machine | $Domain), $User,  
$Attribute=>$Value[ ,  
$Attribute2=>$Value2 ] ));
```

The first parameter (`$Machine` or `$Domain`) is the proper machine or domain name from which you want to get the list of groups. If you specify a domain name, it will use either the PDC or a BDC (whichever is most readily available). If you need to be certain that the list comes from the PDC, you need to pass in the proper name of the PDC. If this is an empty string, a domain controller for the domain currently logged on to will be used.

The second parameter (`$User`) is the name of the user account.

The third and fourth parameters (`$Attribute=>$Value` and `$Attribute2=>$Value2`) are the attribute and attribute's value, respectively. The list of attributes is found in [Table 3.5](#).

Optional fifth and sixth parameters are just like the third and fourth parameters. These optional parameters enable you to set all attributes with one call to this function.

If `UserSetInfo()` is successful, it returns `TRUE`; otherwise, it returns `FALSE`.

**Table 3.5. User Attributes Used with Both `UserGetInfo()` and `UserSetInfo()` from the `Win32::RASAdmin` Extension**

Attribute	Description
<code>Callback</code>	If the callback privilege is set, the RAS server will call the user back using this phone number. There is no limit to the number. It can be a long-distance or local call. Any valid phone number syntax is accepted.
<code>Privilege</code>	A set of flags that describe the user's privileges. The list of values is a combination of values from <a href="#">Table 3.6</a> .

**Table 3.6. RAS Privileges**

Privilege	Description
<code>RASPRIV_DialinPrivilege</code>	Sets the account so that the user is allowed to dial in to a RAS server. If an account does not have this flag set, that user cannot log on to a RAS server.
<code>RASPRIV_NoCallback</code>	Specifies that there is no callback phone number. After the user dials in and is authenticated, he is then online.
<code>RASPRIV_AdminSetCallback</code>	Specifies that only the administrator can set the callback phone number. This number can be set using the <code>Win32::RasAdmin::UserSetInfo()</code> function.
<code>RASPRIV_CallerSetCallback</code>	Specifies that the user can determine which number the server will use as the callback number. When the user logs on, he will be prompted for a callback phone number.

## Group Account Management

Where you have user accounts, you will find groups. Basically speaking, a group is a collection of user accounts. You might want to group all users who have administrative privileges into a group called Admins. Anyone who is a guest could be placed into a Guests group. You could also have groups for Email Users and Accountants. In this respect, an NT group is very similar to a traditional UNIX group.

The beauty of NT groups is that a user can be in multiple groups simultaneously. You could have an accountant with email access, so her account could be in both the Accountants and Email Users groups.

You need to know about two types of groups: global groups and local groups.

A *global* group can contain only user accounts and can be accessed from any machine that participates in the domain. You might put all administrators into your global Domain Admins group, for example. This way, all the machines in your domain can see and use this global group. You can configure each workstation to allow the Domain Admins total unrestricted access over it. If one of your administrators leaves, you can just remove his name from the Domain Admins global group so that he does not have access over any of the machines in the domain.

A *local* group is a group defined on a particular machine, which can contain both user accounts and global groups. Unlike global groups, a local group can only be accessed from the workstation that defines the group. You can add the domain's global Domain Admins group into the local Administrators group on each of your workstations so that each machine allows all members of the Domain Admins group administrative access over the machine.

Each group must have a unique name. No two groups can be named the same even though one might be global and the other local. The only exception is local groups defined on different machines.

### Note

*Global and local groups are not the same as global and local user accounts. The terms "global" and "local" have different meanings when used for groups than they have when used for user accounts.*

*A full description of these meanings is beyond the scope of this book, but it is important to know the differences that exist between the use of "global" versus "local." A good NT/Windows 2000/XP administrator's book is recommended.*

Almost all the group functions that appear in the Win32 extensions come in two flavors: one for local groups and another for global groups. Ideally, this would be hidden from the Perl coder by using one function that determines whether the specified group is local or global, but this is not the case. The rationale for having two functions for each type of group comes from the Win32 API, which specifies two variations for every group function. It would appear that the original group management functions were direct interfaces to these Win32 API functions. All this means is that you might need to try both functions to get an accurate answer. If you are testing to see whether a user is a member of a group, for example, you need to first test by checking the local group; if that fails, you then check the global group. Because there is typically no way to determine whether a given group is local or global, you should indeed test for both possibilities.

## Creating Groups

You can create groups by using one of two functions:

```
Win32::NetAdmin::GroupCreate( $Machine, $Name, $Comment );
Win32::NetAdmin::LocalGroupCreate( $Machine, $Name, $Comment );
```

Both functions are easy to use. You just pass in the proper machine name, the name of the new group, and a comment. If the call is successful, a `true` value is returned; otherwise, `false` is returned.

The first parameter (`$Machine`) can be any valid machine name or an empty string. If it is a valid machine name, the group will be added to that machine if you have permissions to add to it. If an empty string ("") is specified, the group will be created on the local machine. If you are adding a global group from a domain, specify the primary domain controller.

The second parameter (`$Name`) is the name of the group. It can be any string up to 256 characters, with the exception that it cannot be an empty string.

The third parameter (`$Comment`) is the group's comment and can be any string up to 256 characters long, including an empty string. [Example 3.5](#) shows how to create a global group.

### ***Example 3.5 Creating a new group***

```
01. use Win32::NetAdmin;
02. my $Machine = "\\\server1";
03. if( Win32::NetAdmin::GroupCreate( $Machine,
04.                               "My Group",
05.                               "This group is a test" ) )
06. {
07.   print "Successful.";
08. }
09. else
10. {
11.   print "Failed: " .
Win32::FormatMessage( Win32::GetLastError() );
12. }
```

## **Adding Users to a Group**

After a group has been created, you need to add users to the group. This is achieved by using the `GroupAddUsers()` function:

```
Win32::NetAdmin::GroupAddUsers( ($Machine | $Domain), $Group,
$User[, $User2[,
... ] ]);  
Win32::NetAdmin::LocalGroupAddUsers( $Machine, $Group, $User[ ,
$User2_[, ... ] ]
);
```

The first parameter (`$Machine | $Domain`) is the proper name of the machine that contains the group. This can also be a proper domain name in the case of a global group. If this is an empty string, the local machine is assumed.

The second parameter (`$Group`) is the name of the group.

The third (and optionally more) parameter (`$User`) is the name of a valid user account. You can specify multiple user accounts by just tacking them to the end of the function's parameter list. You could just use an array of user accounts for this parameter.

If successful, this function returns a `TRUE` value; otherwise, it returns `FALSE`.

Notice that [Example 3.6](#) first tries to retrieve a list of users from the group. This is done to make sure that the group exists before attempting to add any users. This way, you can also determine whether the group is a local or global group.

### ***Example 3.6 Adding users to a group***

```
01. use Win32::NetAdmin;
02. @Users = qw(
03.     Patrick
04.     Jonathan
05.     William
06.     Leonard
07. );
08. my $Domain = "Staff";
09. my $Group = "Crew";
10. my $Server = "";
11. my $Comment;
12. Win32::NetAdmin::GetDomainController( '', $Domain, $Server );
13. # Let's check to see if the group exists and if it does then
we
14. # will know what type of group it is.
15. if( Win32::NetAdmin::GroupGetAttributes( $Server, $Group,
$Comment ) )
16. {
17.     $Result = Win32::NetAdmin::GroupAddUsers( $Server, $Group,
 \@Users );
18. }
19. else
20. {
21.     if( Win32::NetAdmin::LocalGroupGetAttributes( $Server,
$Group,
$Comment ) )
22.     {
23.         $Result = Win32::NetAdmin::LocalGroupAddUsers( $Server,
$Group,
$Comment );
24.     }
25.     else
26.     {
27.         die "The group $Group does not exist.\n";
28.     }
29. }
30. else
31. {
32.     print "Users have been added to the group $Group.\n";
33. }
34. if( $Result )
35. {
36.     print "Users have been added to the group $Group.\n";
37. }
38. else
```

```
39. {
40.     print "Could not add users to the group $Group.\n";
41. }
```

## Removing Users from a Group

Removing users from a group is very similar to adding users to a group. To remove users from a group, you can use the global or local form of the `GroupDeleteUsers()` function:

```
Win32::NetAdmin::GroupDeleteUsers( ($Machine | $Domain), $Group,
$User[, $User2[,
... ] ] );
Win32::NetAdmin::LocalGroupDeleteUsers( $Machine, $Group, $User[, $User2 [, ... ]
] );
```

The first parameter (`$Machine | $Domain`) is the proper name of the machine that contains the group. This can also be a proper domain name in the case of a global group. If this is an empty string, the local machine is assumed.

The second parameter (`$Group`) is the name of the group.

The third (and optionally more) parameter (`$User`) is the name of a valid user account. You can specify multiple user accounts by just tacking them to the end of the function's parameter list. You could just use an array of user accounts for this parameter.

If successful, this function returns a `TRUE` value; otherwise, it returns `FALSE`.

## Removing a Group

To remove a group, you need to know what kind of group it is: local or global. Two functions accommodate the removal:

```
Win32::NetAdmin::GroupDelete( ($Machine | $Domain), $Group );
Win32::NetAdmin::LocalGroupDelete( $Machine, $Group );
```

Just like the functions that add a group, the first parameter (`$Machine`) is a valid machine name (in the form of '`\machine`') or an empty string ("") that indicates the local machine. If you are removing a global group from a domain, specify the primary domain controller.

The second parameter (`$Group`) is the name of the group to be deleted.

If successful, the group will be removed from the machine, and a `TRUE` value will be returned; otherwise, a `FALSE` value will be returned.

The code in [Example 3.7](#) defines a subroutine that accepts a machine name and group name. This example also attempts to remove the group regardless of whether it is global or local.

The `GroupDelete()` and `LocalGroupDelete()` functions will return a `TRUE` value if successful and a `FALSE` value if they fail.

### **Example 3.7 Removing a group name**

```
01. use Win32::NetAdmin;
02. if( MyDeleteGroup( "\\\server1", "My Group" ) )
03. {
04.     print "Successful!\n";
05. }
06. else
07. {
08.     print "Failed: " .
Win32::FormatMessage( Win32::GetLastError() );
09. }
10.
11. sub MyDeleteGroup
12. {
13.     my( $Machine, $Group ) = @_;
14.     my( $Result ) = 1;
15.     if( ! Win32::NetAdmin::GroupDelete( $Machine, "My Group" ) )
16.     {
17.         if( ! Win32::NetAdmin::LocalGroupDelete( $Machine, "My
Group" ) )
18.         {
19.             $Result = 0;
20.         }
21.     }
22.     return $Result;
23. }
```

## **Retrieving Lists of Groups**

If you need to discover which groups exist, you can use the `GetGroups()` function:

```
Win32::AdminMisc::GetGroups( ($Machine | $Domain), $GroupType,
(\@List | \%List)
[, $Prefix ] ));
```

The first parameter (`$Machine | $Domain`) is the machine from which you want to get the list of groups. This can also be a domain name. If you specify a domain name, it will use either the PDC or a BDC (whichever is most quickly available). If you need to be certain that the list comes from the PDC, you must pass in the name of the PDC. If this is an empty string, a domain controller for the domain currently logged on to will be used.

The second parameter (`$GroupType`) is the type of group for which you are looking. [Table 3.7](#) lists possible values.

**Table 3.7. Group Types to be Used With `Win32::AdminMisc::GetGroups()`**

Group Type	Description
GROUP_TYPE_LOCAL	Retrieves the list of local groups
GROUP_TYPE_GLOBAL	Retrieves the list of global groups
GROUP_TYPE_ALL	Retrieves all group names (both local and global)

The third parameter for the `GetGroups()` function (`\@List` or `\%List`) is a reference to an array or a hash that will be populated with a list of group names. If a hash is specified, it will be populated with subhashes of group information.

The fourth parameter (`$Prefix`) is optional. If this is specified, only group names that begin with the characters specified in this parameter are returned. Suppose you need to collect only the group names that begin with "Test" (your test groups). You would specify "Test" as this parameter. This is convenient if your domain has thousands of groups and you need to access only a few of them.

Because of restrictions of the Win32 API, you can only retrieve the list of local groups if you hold administrative privileges (such as administrators and account operators). Anyone can retrieve the global group list. The `GetGroups()` function will populate the specified array or hash and will return a value of `TRUE` if successful and a value of `FALSE` if not. [Example 3.8](#) shows the `GetGroups()` function in action.

### ***Example 3.8 Retrieving the list of group names***

```

01. use Win32::AdminMisc;
02. my $Domain = "Accounting";
03. if( Win32::AdminMisc::GetGroups( $Domain, GROUP_TYPE_LOCAL,
\@Groups ) )
04. {
05.   DumpGroups( "local groups", @Groups );
06. }
07. if( Win32::AdminMisc::GetGroups( $Domain, GROUP_TYPE_GLOBAL,
\@Groups ) )
08. {
09.   DumpGroups( "global groups", @Groups );
10. }
11.
12. sub DumpGroups
13. {
14.   my( $Type, @List ) = @_;
15.   my( $iCount ) = 0;
16.   print "This is the list of $Type:\n";
17.   map { printf( "\t%03d) %s\n", ++$iCount, $_ ); } @List;
18. }
```

## **Managing Group Attributes**

After a group has been created, you can get/set its comment by using the `GroupGetAttributes()`, `LocalGroupGetAttributes()`, `GroupSetAttributes()`, and `LocalGroupSetAttributes()` functions, the syntax for which is as follows:

```

Win32::NetAdmin::GroupGetAttributes( ($Machine | $Domain), $Group,
$Comment );
Win32::NetAdmin::LocalGroupGetAttributes( $Machine, $Group,
$Comment );
Win32::NetAdmin::GroupSetAttributes( ($Machine | $Domain), $Group,
$Comment );
Win32::NetAdmin::LocalGroupSetAttributes( $Machine, $Group,
$Comment );

```

All four of these functions accept the same three parameters.

The first parameter (`$Machine | $Domain`) is the name of the machine that defines the group. An empty string indicates the local machine. To access a domain's global group, specify the primary domain controller.

The second parameter (`$Group`) is the name of the group.

The third parameter (`$Comment`) is the comment for the group.

The `GroupSetAttributes()` and `LocalGroupSetAttributes()` functions will change the comment of the group, whereas the `GroupGetAttributes()` and `LocalGroupGetAttributes()` functions will retrieve the comment for the specified group and assign it to the third variable.

If the function used is successful, it will return a `true` value; otherwise, it returns `false`.

[Example 3.9](#) demonstrates the functions for retrieving the comment for a given group.

### **Example 3.9 Retrieving a group's comment**

```

01. use Win32::NetAdmin;
02. my $Domain = "Accounting";
03. my $Group = "Domain Admins";
04. my $GroupFound = 1;
05. my $Server = "";
06. Win32::NetAdmin::GetDomainController( '', $Domain, $Server );
07. if( ! Win32::NetAdmin::LocalGroupGetAttributes( $Server,
$Group, $Comment ) )
08. {
09.   if( ! Win32::NetAdmin::GroupGetAttributes( $Server, $Group,
$Comment ) )
10.   {
11.     $GroupFound = 0;
12.   }
13. }
14. if( $GroupFound )
15. {
16.   print "The comment for $Group is: '$Comment'.\n";
17. }
18. else
19. {
20.   print "The group $Group could not be found.\n";

```

```
21. }
```

## **Listing Users in a Group**

Quite often, you need to process a list of users who are associated with some group. If you have a group called Email Users, for example, you might need to change its logon scripts. To do this, you would first need to know exactly who is in the Email Users group. This is where the global and local versions of the `GroupGetMembers()` function come into play:

```
Win32::NetAdmin::GroupGetMembers( ($Machine | $Domain), $GroupName,  
 \@Users );  
Win32::NetAdmin::LocalGroupGetMembers( $Machine, $GroupName,  
 \@Users );  
Win32::NetAdmin::LocalGroupGetMembersWithDomain( $Machine,  
 $GroupName, \@Users );
```

There is no difference between the first two functions other than one only works with global groups and the other with local groups. The third function is available from the core distribution's Win32 library version 0.12 (`libwin32-0.12`) and later versions. It retrieves both the domain and the username or each account in a local group.

The first parameter (`$Machine | $Domain`) is the name of the machine that holds the list. If this is an empty string, the local machine is assumed.

The second parameter (`$GroupName`) is the name of the group.

The third parameter (`\@Users`) is a reference to an array that will be populated with the user accounts that are members of the group. If using the `LocalGroupGetMembersWithDomain()` function, the array will be populated with the members' usernames and their domains in the form of '`domain\ user-name`'.

Both functions will return a `TRUE` value if successful; otherwise, they will return `FALSE`.

[Example 3.10](#) uses the `GroupGetMembers()` function to change the logon script for all members in a particular group. Notice that the code tries both versions of the function because it does not know whether the group is a local or global group.

### ***Example 3.10 Retrieving the members in a group***

```
01. use Win32::NetAdmin;  
02. use Win32::AdminMisc;  
03. my $Group = "Email Users";  
04. my $Domain = "Accounting";  
05. my $LogonScript = "EmailUsers.pl";  
06. my $Server = "";  
07. my @UserList;  
08. Win32::NetAdmin::GetDomainController( '', $Domain, $Server );  
09. # Get the list of group members  
10. if( ! Win32::NetAdmin::GroupGetMembers( $Server, $Group,  
 \@UserList ) ) |
```

```

11. {
12. Win32::NetAdmin::LocalGroupGetMembers( $Server, $Group,
13. \@UserList ) ||
14.     die "There is no group called '$Group'.\n";
15. }
16. {
17.     if( Win32::AdminMisc::UserSetMiscAttributes( $Server, $User,
18.                                                 USER_LOGON_SCRIPT=>$LogonScript ) )
19.     {
20.         print "Successfully changed logon script for user
21. '$User'.\n";
22.     }

```

## Verifying User Membership in a Group

Sometimes it is quicker to just test and see whether a particular user is a member of a group as opposed to retrieving the list of members and searching through it. You can use the `GroupIsMember()` function for this:

```

Win32::NetAdmin::GroupIsMember( ($Machine | $Domain), $GroupName,
$User );
Win32::NetAdmin::LocalGroupIsMember( $Machine, $GroupName, $User );

```

The first parameter (`$Machine | $Domain`) is the proper name of the machine or domain that houses the group. If this is an empty string, the local machine is assumed.

The second parameter (`$GroupName`) is the name of the group.

The third parameter (`$User`) is the user account to be checked. If these functions are successful, they return `TRUE`; otherwise, they return

`FALSE`.

## Machine Management

Administrating a domain of machines can be quite a job. Users are always deleting system files, changing bootup configurations, and reconfiguring software. Something is always happening to keep a team of administrators running from desktop to desktop. Luckily, Perl's Win32 extensions have some handy functions that make this administration easier.

Administrators of Win32 machines need to manage INI files and the Registry. Additionally, NT/Win2k/XP machines need the Event Log to be managed. Perl provides the capability to perform such tasks.

## Discovering Drives

You can find what drives a computer has by calling the `Win32::Lanman::NetServerDiskEnum()` function:

```
Win32::Lanman::NetServerDiskEnum( $Machine, \@List );
```

The first parameter (`$Machine`) is the name of a machine you want to query. If this is an empty value (either `undef` or `" "`), the local machine is examined.

The second parameter (`\@List`) is an array reference. The list of drives attached to the machine are added to the end of the array. Each drive is indicated by its drive letter (such as A:, C:, and S:).

This function does not clear the array passed in. Therefore, unless you empty the array before calling the function, it might be difficult to determine what drives were reported by the function.

Alternatively, you can call the `Win32::AdminMisc::GetDrives()` function to discover what drives you have on your local machine:

```
Win32::AdminMisc::GetDrives( [$DriveType] );
```

The only parameter is optional and can be any value from [Table 3.8](#). If the parameter is not specified, *all* drives are returned.

If the function is successful, it returns an array of drive letters (such as A:, C:, and so on). Each drive letter represents a drive that matches the specified type.

**Table 3.8. Different Drive Types**

Constant	Description
<code>DRIVE_REMOVABLE</code>	The drive is removable. This could be a floppy, a zip drive, a Jaz drive, and so on.
<code>DRIVE_FIXED</code>	The drive is a hard disk.
<code>DRIVE_REMOTE</code>	The drive is a UNC mapped to a drive letter.
<code>DRIVE_CDROM</code>	The drive is a CD-ROM.
<code>DRIVE_RAMDISK</code>	The drive is a RAM disk.

## Machine Configurations

Every Win32 machine can act as both a workstation and a service. That is, it can connect to other machines (acting as a workstation) and allow other machines to connect to it (acting as a server). How a machine works totally depends on what services are running. By default, an NT/2000/XP machine enables the workstation and server services, thus allowing both functionalities.

Each of these capabilities is configurable. This gives an administrator, who knows what he is doing, the capability to tweak settings to maximize performance for his particular network topology.

## Warning

*Modifying a computer's workstation or server settings can lead to configuration problems with the machine. Be advised that you should not try to modify any of a computer's server configuration parameters unless you are certain that you know what you are doing. This could lead to serious performance impacts, among other problems.*

## Configuring Workstations

The `Win32::Lanman` extension provides functions to query and modify a machine's workstation settings. These are the `NetWkstaGetInfo()` and `NetWkstaSetInfo()` functions:

```
Win32::Lanman::NetWkstaGetInfo( $Machine, \%Info [, $bFullInfo]);  
Win32::Lanman::NetWkstaSetInfo( $Machine, \%Info );
```

The first parameter (`$Machine`) is the proper machine name of the computer whose workstation is to be queried or configured. If this is `undef` or an empty string, the local machine is used.

The second parameter (`\%Info`) must be a reference to a hash. Either this will be populated with configuration information (in the case of `NetWkstaGetInfo()`), or its values will be used to configure the workstation (in the case of `NetWkstaSetInfo()`).

The optional third parameter (`$bExtendedInfo`) is a Boolean value that determines what information the second parameter specifies. If this value is not specified or is `0`, the hash represents the configuration settings in [Table 3.9](#). If this parameter is any nonzero value, the hash represents configuration settings from both [Table 3.9](#) and [Table 3.10](#).

The function returns `TRUE (1)` if the function is successful; otherwise, it returns `FALSE (0)`.

Note that not all settings can be modified. Only the settings in [Table 3.10](#) that are *not* marked as read-only can be modified. Unfortunately, the settings from [Table 3.9](#) cannot be modified.

**Table 3.9. Basic Workstation Configuration Information**

Key Name	Description
<code>computername</code>	The name of the machine.
<code>langroup</code>	The domain name to which the computer belongs.
<code>lanroot</code>	The path (on the machine) that points to the MS LAN Manager directory. This typically is an empty string on NT/Win2k/XP machines and is provided for legacy support to Microsoft's LAN Manager product.
<code>logged_on_users</code>	A numeric value indicating the total number of user accounts that are logged on to the machine.

**Table 3.9. Basic Workstation Configuration Information**

Key Name	Description
platform_id	Indicates the platform on which the server runs. Valid values are:  PLATFORM_ID_DOS  PLATFORM_ID_OS2  PLATFORM_ID_NT  PLATFORM_ID_OSF  PLATFORM_ID_VMS
version_major	The major version of the OS. For an NT 3.51 server, this value would be 3. Note that only the least significant 4 bits really indicate the major version. Use a Boolean AND of this key with MAJOR_VERSION_MASK to get the true value.
version_minor	The minor version of the OS. For an NT 3.51 server, this value would be 51.

**Table 3.10. Extended Workstation Configuration Information**

Key Name	Description
char_wait	Specifies the number of seconds the computer waits for a remote resource to become available.
collection_time	Specifies the number of milliseconds during which the computer collects data before sending the data to a character device resource. The workstation waits the specified time or collects the number of characters specified by the maximum_collection_count property, whichever comes first.
maximum_collection_count	Specifies the number of bytes of information the computer collects before sending the data to a character device resource. The workstation collects the specified number of bytes or waits the period of time specified by the collection_time property, whichever comes first.
keep_conn	Specifies the number of seconds the server maintains an inactive connection to a server's resource.
max_cmds	Specifies the number of simultaneous network device driver commands that can be sent to the network.
	This is a read-only value.

**Table 3.10. Extended Workstation Configuration Information**

Key Name	Description
<code>sess_timeout</code>	Indicates the number of seconds the server waits before disconnecting an inactive session.
<code>siz_char_buf</code>	Specifies the maximum size, in bytes, of a character pipe buffer and device buffer.
<code>max_threads</code>	Specifies the number of threads the computer can dedicate to the network.
<code>lock_quota</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
<code>lock_increment</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
<code>lock_maximum</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
<code>pipe_increment</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
<code>pipe_maximum</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
<code>cache_file_timeout</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
<code>dormant_file_limit</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
<code>read_ahead_throughput</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
<code>num_mailslot_buffers</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
	This is a read-only value.
<code>num_srv_announce_buffers</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
	This is a read-only value.
<code>max_illegal_datagram_events</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
	This is a read-only value.
<code>illegal_datagram_event_reset_frequency</code>	<i>This value is not described by Microsoft's Win32 API documentation.</i>
	This is a read-only value.

**Table 3.10. Extended Workstation Configuration Information**

Key Name	Description
log_election_packets	<i>This value is not described by Microsoft's Win32 API documentation.</i>
use_opportunistic_locking	This is a read-only value. <i>This value is not described by Microsoft's Win32 API documentation.</i>
use_unlock_behind	<i>This value is not described by Microsoft's Win32 API documentation.</i>
use_close_behind	<i>This value is not described by Microsoft's Win32 API documentation.</i>
buf_named_pipes	<i>This value is not described by Microsoft's Win32 API documentation.</i>
use_lock_read_unlock	<i>This value is not described by Microsoft's Win32 API documentation.</i>
utilize_nt_caching	<i>This value is not described by Microsoft's Win32 API documentation.</i>
use_raw_read	<i>This value is not described by Microsoft's Win32 API documentation.</i>
use_raw_write	<i>This value is not described by Microsoft's Win32 API documentation.</i>
use_write_raw_data	<i>This value is not described by Microsoft's Win32 API documentation.</i>
use_encryption	<i>This value is not described by Microsoft's Win32 API documentation.</i>
buf_files_deny_write	<i>This value is not described by Microsoft's Win32 API documentation.</i>
buf_read_only_files	<i>This value is not described by Microsoft's Win32 API documentation.</i>
force_core_create_mode	<i>This value is not described by Microsoft's Win32 API documentation.</i>
use_512_byte_max_transfer	<i>This value is not described by Microsoft's Win32 API documentation.</i>

## Configuring Servers

By using `Win32::Lanman`, you can both discover and modify the way a server works. The `NetServerGetInfo()` and `NetServerSetInfo()` functions help do this:

```

Win32::Lanman::NetServerGetInfo( $Server, \%Info [,  

$bExtendedInfo] );  

Win32::Lanman::NetServerSetInfo( $Server, \%Info [,  

$bExtendedInfo] );

```

The first parameter (`$Server`) is the name of the server. This must be a proper computer name; therefore, it must be prefixed with double backslashes. If no value is specified (for example, if `undef` is passed in), the local computer will be used.

The second parameter (`\%Info`) must be a reference to a hash. Either this will be populated with configuration information (in the case of `NetServerGetInfo()`), or its values will be used to configure the server (in the case of `NetServerSetInfo()`).

The optional third parameter (`$bExtendedInfo`) is a Boolean value that determines what information the second parameter specifies. If this value is not specified or is `0`, the hash represents the configuration settings in [Table 3.11](#). If this parameter is any nonzero value, the hash represents configuration settings from both [Table 3.11](#) and [Table 3.12](#).

The function returns `true` (1) if the function is successful; otherwise, it returns `false` (0).

**Table 3.11. Basic Server Configuration Information**

Key Name	Description
<code>platform_id</code>	Indicates the platform on which the server runs. Valid values are:  <code>PLATFORM_ID_DOS</code>  <code>PLATFORM_ID_OS2</code>  <code>PLATFORM_ID_NT</code>  <code>PLATFORM_ID_OSF</code>  <code>PLATFORM_ID_VMS</code>
<code>name</code>	The name of the server.
<code>version_major</code>	The major version of the OS. For an NT 3.51 server, this value would be 3. Note that only the least significant 4 bits really indicate the major version. Use a Boolean AND of this key with <code>MAJOR_VERSION_MASK</code> to get the true value.
<code>version_minor</code>	The minor version of the OS. For an NT 3.51 server, this value would be 51.
<code>type</code>	The platform on which the server is running. This can be any one value from <a href="#">Table 2.1</a> (see <a href="#">Chapter 2</a> ).
<code>comment</code>	The server's comment.
<code>users</code>	A numeric value indicating the total number of users allowed to simultaneously log on to the server.
<code>disc</code>	The number of minutes the server waits before disconnecting idle users. If this

**Table 3.11. Basic Server Configuration Information**

Key Name	Description
hidden	value is <code>SV_NODISC</code> , no disconnection will occur.
announce	This determines whether the server is not visible on the network through browsing services. This value can be one of the following:
	<code>SV_VISIBLE</code> (The server is visible.)
	<code>SV_HIDDEN</code> (The server is not visible.)
anndelta	This identifies how often the server announces its presence on the network. This value is in seconds.
	This is the amount of time that network announcements are allowed to be varied. This value is in milliseconds. How often the server will be announced to the network is determined by:
	<code>announce ± anndelta</code>
licenses	Specifies the number of users per license. By default, this number is <code>SV_USERS_PER_LICENSE</code> .
userpath	The path to a user's directories on the server.

**Table 3.12. Extended Server Configuration Information**

Key Name	Description
<code>sessopens</code>	The number of files that can be open in any one session.
<code>sessvcs</code>	The maximum number of "virtual circuits" permitted per client.
<code>opensearch</code>	The number of search operations that can occur simultaneously.
<code>sizreqbuf</code>	The size of each server buffer. This value is in bytes.
<code>initworkitems</code>	The initial number of work items (a.k.a. "receive buffers") that the server uses.
<code>maxworkitems</code>	Specifies the maximum number of receive buffers, or work items, that the server can allocate. If this limit is reached, the transport must initiate flow control at a significant performance cost.
<code>rawworkitems</code>	Specifies the number of special work items the server uses for raw mode I/O. A larger value for this member can increase performance, but it requires more memory.
<code>irpstacksize</code>	Specifies the number of stack locations that the server allocated in I/O request packets (IRPs).
<code>maxrawbuflen</code>	Specifies the maximum raw mode buffer size. This value is in bytes.
<code>sessusers</code>	Specifies the maximum number of users that can be logged on to the

**Table 3.12. Extended Server Configuration Information**

<b>Key Name</b>	<b>Description</b>
sessconns	server using a single virtual circuit.
maxpagedmemoryusage	Specifies the maximum size of pageable memory that the server can allocate at any one time.
maxnonpagedmemoryusage	Specifies the maximum size of nonpaged memory that the server can allocate at any time.
enablessoftcompat	Enables software compatibility. <i>This value is not described by Microsoft's Win32 API documentation.</i>
enableforcedlogoff	Specifies whether the server should force a client to disconnect, even if the client has open files, when the client's logon time has expired.
timesource	Specifies whether the server is a reliable time source.
acceptdownlevelapis	Specifies whether the server accepts function calls from previous-generation LAN Manager clients.
lmannounce	Specifies whether the server is visible to LAN Manager 2.x clients.
domain	The name of the domain in which the server participates.
maxcopyreadlen	<i>This value is not described by Microsoft's Win32 API documentation.</i>
maxcopywritelen	<i>This value is not described by Microsoft's Win32 API documentation.</i>
minkeepsearch	<i>This value is not described by Microsoft's Win32 API documentation.</i>
maxkeepsearch	Specifies the length of time that the server retains information about incomplete search operations.
minkeepcomplsearch	<i>This value is not described by Microsoft's Win32 API documentation.</i>
maxkeepcomplsearch	<i>This value is not described by Microsoft's Win32 API documentation.</i>
threadcountadd	<i>This value is not described by Microsoft's Win32 API documentation.</i>
numblockthreads	<i>This value is not described by Microsoft's Win32 API documentation.</i>
scavtimeout	Specifies the period of time that the scavenger remains idle before waking up to service requests.
minrcvqueue	Specifies the minimum number of free receive work items the server requires before it begins to allocate more.
minfreeworkitems	Specifies the minimum number of available receive work items that the server requires to begin processing a server message block.
xactmemsize	Specifies the size of the shared memory region used to process server functions.
threadpriority	Specifies the priority of all server threads in relation to the base priority of the process.

**Table 3.12. Extended Server Configuration Information**

Key Name	Description
maxmpxct	Specifies the maximum number of outstanding requests that any one client can send to the server. For example, 10 means you can have 10 unanswered requests at the server. When any single client has 10 requests queued within the server, the client must wait for a server response before sending another request.
oplockbreakwait	Specifies the period of time to wait before timing out an opportunistic lock break request.
oplockbreakresponsewait	<i>This value is not described by Microsoft's Win32 API documentation.</i>
enableoplocks	Specifies whether the server allows clients to use opportunistic locks on files. Opportunistic locks are a significant performance enhancement but have the potential to cause lost cached data on some networks, particularly wide area networks.
enableoplockforceclose	<i>This value is not described by Microsoft's Win32 API documentation.</i>
enablefcopens	Specifies whether several MS-DOS File Control Blocks (FCBs) are placed in a single location accessible to the server. If enabled, this can save resources on the server.
enableraw	Specifies whether the server processes raw Server Message Blocks (SMBs). If enabled, this allows more data to transfer per transaction and also improves performance. However, it is possible that processing raw SMBs can impede performance on certain networks. The server maintains the value of this member.
enablesharednetdrives	Specifies whether the server allows redirected server drives to be shared.
minfreeconnections	Specifies the minimum number of free connection blocks maintained per endpoint. The server sets these aside to handle bursts of requests by clients to <code>connect to the server</code> .
maxfreeconnections	Specifies the maximum number of free connection blocks maintained per endpoint. The server sets these aside to handle bursts of requests by clients to connect to the server.

## Managing INI Files

For many years now, INI files have plagued administrators because they are so easy for any user to change. Because there are no tools in NT/Windows 2000/XP or Windows 95/98/ME (let alone Win 3.x and DOS) to manage INI files, administrators have been using Perl to do the work.

Programs use INI files to hold configuration information. Think of INI as standing for INIInitialization, as in the configuration used to initialize a program. These files follow a particular format, which makes it easy to manage because all INI files follow this format. The file is broken into sections, each having a unique name. Each section can contain several keys that have equated values.

[Example 3.11](#) shows an INI file that contains one section called `FileManager`. (This was taken

from a Windows 95 machine's File Manager INI file.) This section has four keys: `Path`, `SearchSpec`, `Flags`, and `SavedSearches`. You can see that the values for these flags can be either numeric or a character string.

### ***Example 3.11 Example of an INI file***

```
[FileManager]
Path=C:\TEMP
SearchSpec=*.doc
Flags= 260
SavedSearches=0
```

Occasionally, you might need to change the contents of an INI file. A company I worked with had to roll out a version of Microsoft's Internet Explorer that worked with NT 3.51. This version of IE (v3.0) made use of an INI file for its configuration. It had been installed on several hundred computers, and the users were using it left and right. A network change was made, and the INI files had to be altered to reflect new proxy settings. A new INI file could have been copied down to everyone's machine during his or her logon script, but that would reset any personalization the user had made. Instead, we made use of the `Win32::AdminMisc` extension's capability to read from and write to INI files, as demonstrated in the following sections.

### ***Reading INI Files***

Data can be read from an INI file by using the `ReadINI()` function:

```
Win32::AdminMisc::ReadINI( $File [, $Section [, $Key]] );
```

The first parameter (`$File`) is the path to an INI file. This does not need to be a full path, but it does need to point to a valid INI file. If only the filename is specified (with no path), the file will be looked for in the current directory. If it is not found there, the Windows directory is searched followed by the Windows System directory and then the environmental variable `%PATH%`.

The second parameter (`$Section`) is the name of the section in the INI file. This can be an empty string.

The third parameter (`$Key`) is the name of a particular key in the specified section. If this parameter is an empty string, the function returns an array of key names.

If the `ReadINI()` function is successful, it returns one of the following:

- The data associated with the key in the specified section from the INI file.
- If the `$Key` parameter is an empty string, an array is returned containing the names of all keys in the specified section.
- If the `$Section` parameter is an empty string, an array is returned containing the names of all sections in the file.

[Example 3.12](#) shows this function at work.

### ***Writing INI Files***

INI files can be altered by using the `WriteINI()` function:

```
Win32::AdminMisc::WriteINI( $File [, $Section [, $Key [, $Value]]] );
```

The first parameter (`$File`) is the path to an INI file. This does not need to be a full path, but it does need to point to a valid INI file. If only the filename is specified (with no path), the file will be looked for in the current directory. If it is not found there, the Windows directory is searched followed by the Windows System directory and then the environmental variable `%PATH%`.

The second parameter (`$Section`) is the name of the section in the INI file. If this is an empty string, the function removes all sections from the specified INI file.

The third parameter (`$Key`) is the name of a particular key in the specified section. If this is an empty string, the function removes all keys in the specified section.

The fourth parameter (`$Value`) is the value that will be associated with the specified key. If this is an empty string, the function removes the specified key.

If the `WriteINI()` function is successful, it returns a `TRUE` value; otherwise, it returns `FALSE`. When the function succeeds, the result will be one of the following:

- The value will be associated with the specified key in the specified section in the INI file. This overwrites any value already present.
- If the value parameter is an empty string, that key will be removed from the section.
- If the key parameter is an empty string, all keys in the specified section will be removed.
- If the section parameter is an empty string, all sections in the file will be removed.

[Example 3.12](#) demonstrates both the `ReadINI()` and `WriteINI()` functions.

### **Example 3.12 Adding and removing data from an INI file**

```
01. use Win32::AdminMisc;
02. my $File = "$ENV{WinDir}\win.ini";
03. my $Section = "Devices";
04. my @Keys = Win32::AdminMisc::ReadINI( $File, $Section, "" );
05. foreach my $Key ( @Keys )
06. {
07.     $Devices{$Key} = Win32::AdminMisc::ReadINI( $File, $Section,
$Key );
08. }
09. # Remove the entire section...
10. if( Win32::AdminMisc::WriteINI( $File, $Section, "", "" ) )
11. {
12.     # Now add the devices again recreating the section...
13.     foreach my $Key ( @Keys )
14.     {
15.         print "$Key=$Devices{$Key}" to [$Section]...\n";
16.         Win32::AdminMisc::WriteINI( $File, $Section, $Key,
$Device{$Key} );
17.     }
```

```
18. }
```

## Tip

*It might be handy to know that `Win32::AdminMisc`'s `ReadINI()` and `WriteINI()` functions are not case sensitive. Therefore, you need not concern your script with matching the case of a section or key name.*

*Additionally, the specified file can use either forward slashes or backslashes. The functions will convert any forward slashes to backslashes before doing its work. This is quite nice if you are using Perl-friendly pathnames as opposed to DOS paths.*

## Tieing an INI File

There is another way to access and manipulate INI files: using the Perl `tie` function. If you are not aware of the `tie` function, it would be best to look it up in a good Perl reference book. This is a very powerful function.

Basically, the `tie` function will associate a variable with a Perl module. Whenever any action is performed on the variable, certain methods in the module are called. When you "tie" a variable to a module, you are adding what is called "magic" to the variable. This makes the variable magic—which is pretty cool!

Consider this example: Suppose you have hooked up your house lights to your computer so that you can control them by means of a program. Assume also that you have written a Perl extension that controls the lights, enabling you to control them via Perl scripts. You could add code to your extension so that a hash, let's call it `%Lights`, is created. Now every time you query the hash, such as `print $Lights{porch}`, the status of the lights returns. If the porch light is on, `$Lights{porch}` returns a `1`; if it is off, it returns a `0`. If you were to set `$Lights{porch}` to `1`, as in `$Lights{porch} = 1`, the porch light would turn on. Likewise, setting it to `0` would turn off the porch light.

This could be done using magic (or tied) hashes. When you tell the hash (`%Lights`) that it is tied to your module, Perl will call your module to handle such things as querying the hashes values, setting the values, enumerating the keys in the hash, and so on. This is what is referred to as tieing or magic (because the hash appears to act as if it were magical).

The `Win32::Tie::Ini` extension provides magic to any hash that is tied with this extension. The hash will be an interface to an INI file.

To use this extension, you must first load it using the `use` command:

```
use Win32::Tie::Ini;
```

After this has been performed, you can tie a hash to any INI file using Perl's `tie` command:

```
tie( %Hash, "Win32::Tie::Ini", $File );
```

The first parameter (`%Hash`) is the hash you want to tie.

The second parameter is the name of the module; in this case, it must be "`Win32::Tie::Ini`".

The third parameter (`$File`) is the name of the INI file. This does not need to be a full path, but it does need to point to a valid INI file. If only the filename is specified (with no path), the file will be looked for in the current directory. If it is not found there, the Windows directory is searched followed by the Windows System directory and then the environmental variable `%PATH%`.

If the `tie` function is successful, it returns a reference to the hash. There is no need to save this reference because it just points to the first parameter, the hash.

After you have successfully tied a hash to an INI file, you can start to read settings. It is important to note that because INI files consist of sections containing keys, the hash will reflect that (in that the hash's key is a section name). The value of a key is a reference to another hash. That other hash will contain the section's keys, which are accessible by means of the hash's keys.

### ***Reading from a Tied INI File***

To read the keys from a section, you must first retrieve a reference to the section's hash, such as:

```
$SectionRef = $Hash{devices};
```

Now `$SectionRef` is a reference to a hash that contains the keys to the section. From this point, you can print out the keys and values of the `[Devices]` section using the following:

```
01. foreach $Key ( keys( %$SectionRef ) )
02. {
03.   print "$Key=$SectionRef->{$Key}\n";
04. }
```

Notice that you must dereference `$SectionRef` to get to its keys because it is a reference to a hash (not just a hash).

You could, of course, dereference the hash reference when you first assign it by using the following:

```
%Section = %{ $Hash{devices} };
```

Now you could use the `%Section` hash as a regular hash:

```
01. foreach $Key ( keys( %Section ) )
02. {
03.   print "$Key=%Section{$Key}\n";
04. }
```

It is very important to note that by dereferencing the hash reference (as in the second example), you have a copy of the INI file's data. Making changes to this new hash will not update the INI file. So if you were to do this:

```
01. %Section = %{$Hash{devices}};  
02. $Section{WinFax} = "This is a test";
```

the `WinFax` key in the `[Devices]` section of the INI file will not be changed. This is because you are changing a key's data in the `%Section` hash. This hash is a copy of the INI data; it is not tied to the INI file at all (as `%Hash` is).

On the other hand, if you used the hash reference as in the first example, changes made to it will be saved to the INI file. This is because the hash reference is still tied to the INI file. For more details about this, consult a Perl manual regarding hashes, hash references, and tieing (such as O'Reilly & Associate's *Advanced Perl Programming*).

### ***Writing to a Tied INI File***

Writing to the INI file is as easy as setting the value of a hash. Continuing from the preceding section, assume that `%Hash` is tied to an INI file (the `WIN.INI` file in particular).

You could set the data for the `WinFax` key in the `[Devices]` section by using the following:

```
$Hash{devices}->{WinFax} = "This is my new test data";
```

It is that easy. You could just as well use this:

```
01. $Section = $Hash{devices};  
02. $Section->{WinFax} = "This is my new test data";
```

Again, notice that only references are used here. If you use a regular hash, it will be a copy of the data from the hash reference—which is not tied, so any changes will not be saved to the INI file.

### ***Removing Data from a Tied INI File***

To remove a key or section in an INI file, all you need to do is use the delete command as you normally would:

```
delete $Hash{devices}->{WinFax};
```

This removes the `WinFax` key from the `[Devices]` section of the INI file. Likewise, the code

```
delete $Hash{devices};
```

removes the entire `[Devices]` section from the INI file.

## **Untying an INI File**

After you finish with the INI file, all you need to do is untie the hash. Then you really are finished! You use Perl's untie command for this:

```
untie %Hash;
```

The code in [Example 3.13](#) demonstrates how easy it is to use the `Win32::Tie::Ini` extension to perform the same task as [Example 3.12](#).

### **Example 3.13 Using the `Win32::Tie::Ini` extension version of Example 3.12**

```
01. use Win32::Tie::Ini;
02. my $File = "$ENV{WinDir}\\\win.ini";
03. my $Section = "Devices";
04. my %Hash;
05. tie( %Hash, "Win::Tie::Ini", $File ) || die "Could not tie to
$file.\n";
06.
07. # Make a *copy* of the INI files section
08. %Devices = %{ $Hash{$Section} };
09.
10. # Remove the entire section...
11. delete $Hash{$Section};
12.
13. # Now add the devices again recreating the section...
14. %{ $Hash{$Section} } = %Devices;
15. untie %Hash;
```

## **Managing the Registry**

Not all configuration information is stored in INI files. Since the Win32 platform was released, Microsoft has been encouraging programmers to make use of the Registry and not use INI files. There is good reason for this: The Registry is a central repository of configuration data. This makes it easier for administrators to find configuration data because it is always in a database rather than scattered among hundreds of directories and other files on a hard drive.

### **Note**

*On Windows NT machines, a user's Registry information can be stored as the user's profile on a file server. If the user's account is configured to use a "roaming profile," when the user logs on to a different machine, it loads the profile from the file server. This enables the user to always have access to his personal configurations, regardless of which machine he might be logged on to.*

On the other hand, in many cases, the Registry becomes so huge with information that it slows the machine down when a program is accessing configuration data from within the Registry. Additionally, it is more difficult to modify the Registry than it is to modify an INI file.

Win32 Perl has a standard Registry extension that provides access to Registry data. This makes administrating these important databases a breeze.

The Registry is a database that consists of keys, subkeys, values, and data. Think of it like a file system. A key is like a directory. Whereas a directory can contain files and other subdirectories, a key can contain values and other subkeys. A value is just a name associated with some type of data, just as a filename is just a name associated to data (the data held within the file).

Just as your Win32 machine can have multiple drives, the Registry has multiple root keys. These roots define what kind of data is held underneath them. To illustrate this, imagine that you have only system files on your C: drive, only data files on your D: drive, and only applications on your E: drive. These roots will come into play in the section on opening keys.

The `Win32::Registry` extension automatically creates `Win32::Registry` objects that point to the Registry's root keys (or Registry hives). [Table 3.13](#) lists these objects.

**Table 3.13. Predefined `Win32::Registry` Objects That Reflect the Win32 Registry Root Keys**

Registry Object	Description
<code>\$HKEY_LOCAL_MACHINE</code>	Points to the hive that contains configuration information about the computer. This includes device drivers and service configurations.
<code>\$HKEY_CURRENT_USER</code>	Points to the current user's hive. This hive is not available when accessing a Registry remotely.
<code>\$HKEY_CLASSES_ROOT</code>	This is the root key for all OLE/COM-related class information. This includes MIME types and file associations. This root is a link to <code>HKEY_LOCAL_MACHINE\Software\Classes</code> . This is important to know because this root key is unavailable when connecting a Registry remotely.
<code>\$HKEY_USERS</code>	This is the root key for all user hives.
<code>\$HKEY_PERFORMANCE_DATA</code>	This is the root key for all performance data.  Win32 does not have a separate API to gather performance data; instead, it uses the Registry API (the <code>RegQueryValueEx()</code> function in particular) to retrieve performance data. The Registry API maps requests using this root key to the actual performance data counters and registers.
<code>\$HKEY_CURRENT_CONFIG</code>	Points to the current configuration in the Registry.
<code>\$HKEY_DYN_DATA</code>	This root key is the Windows 95/98/ME equivalent to the NT/Win2k/XP <code>HKEY_PERFORMANCE_DATA</code> hive.

A hive is any key and all its values and subkeys. Typically, a hive will represent a tree of keys and subkeys related to something. A user's personal configuration on an NT machine is stored as part of the user's profile, for example. When the user logs on to a machine, it will load up that user's hive into the Registry and point the `HKEY_CURRENT_USER` root key to it. If a user's profile does not exist,

a new one is created based on the default profile. This way, you always know that the current user's configuration information is available via that root key. A hive is basically any collection of keys, subkeys, and values.

Hives can be loaded into and unloaded from the Registry. An administrator might need to modify a particular user's Registry settings, for example. The administrator could log on to a machine using the user's ID and password (presuming that he knew it or reset it). This would cause the user's personal hive to be loaded into the Registry and linked to the `HKEY_CURRENT_USER` root. The administrator could then proceed to edit the `HKEY_CURRENT_USER`'s contents. When the administrator logs off, the hive is unloaded from the Registry. This could be quite a burden if the administrator had to change several users' configurations.

The alternative to this tedious approach is to manually load the user's hive into the Registry of some machine. When a hive is manually loaded, a temporary key name is specified and created, such as Joel's Hive. If the hive loaded successfully, the administrator can edit the contents of the root key Joel's Hive. When finished, the hive is just unloaded. This technique enables a Perl script to systematically load, modify, and unload any number of hives quickly and without human interaction.

## Note

*The authors of the `Win32::Registry` extension included some pretty useful Registry functions. For some unknown reason, however, they did not provide access to all these methods through its module. This means that, to use these functions, you have to access the extension's functions directly as functions, not as methods. Each function will be discussed when appropriate.*

## Opening a Registry Key

When accessing data in the Registry, you need to open a key, just like you would open a directory using the `opendir()` function. After the key has been opened, you can manipulate the values and data within the key. To open a key, use the `Open()` method:

```
$RegistryObject->Open( $Path, $Key )
```

Notice that `Open()` is a method of a `Win32::Registry` object. The first time you are opening a Registry key, you should use one of the root key objects (which are predefined by the extension) listed in [Table 3.9](#).

The first parameter (`$Path`) passed into the method is the path to the key to be opened. The path is relative to the object calling the method. If you want to open `HKEY_LOCAL_MACHINE\Software\ActiveState`, for example, you specify a path of `Software\ActiveState`, and the method is called from the `$HKEY_LOCAL_MACHINE` object.

The second parameter (`$Key`) is a scalar that will become a Registry object. You will use this later to call other methods.

If successful, the `Open()` method returns a `true` value, a new Registry object is created, and the scalar specified as the second parameter is set to the new object. If the method fails, it returns `false`.

The method might fail because the key does not exist or because you do not have permissions to open the key.

Now that a key is opened, you can enumerate subkeys and values as well as retrieve and set data. See [Example 3.14](#).

### ***Example 3.14 Opening a Registry key***

```
01. use Win32::Registry;
02. my $Key;
03. if( $HKEY_LOCAL_MACHINE->Open( "Software\\ActiveState" ,
$Key ) )
04. {
05.     #...process registry key...
06.
07.     $Key->Close();
08. }
09. else
10. {
11.     print "Could not open the key: $^E\n";
12. }
```

### ***Creating a Registry Key***

Just like opening a key, you can create a key using the `Create()` method. Creating a key not only will create the key, it will open it just like the `Open()` method. If the key already exists, the key is opened and the method is successful.

```
$RegistryObject->Create( $Path, $Key );
```

The first parameter passed into the `Create()` method (`$Path`) is the path to the key to be created. The path is relative to the object calling the method. If you want to create `HKEY_LOCAL_MACHINE\Software\ActiveState\Perl5`, for example, you would specify a path of `Software\ActiveState\Perl5`, and the method is called from the `$HKEY_LOCAL_MACHINE` object.

The second parameter (`$Key`) is a scalar variable that will become a Registry object. You will use this later to call other methods.

If successful, the `Create()` method returns a `TRUE` value, the specified key and a new Registry object are created, and the scalar specified as the second parameter is set to the new object. If the `Create()` method fails, it returns `FALSE`. The `Create()` method might fail because the key was unable to be created or opened due to permissions or delete/change operations being performed on any of its parent keys. [Example 3.15](#) demonstrates this method.

### ***Example 3.15 Creating a Registry key***

```
01. use Win32::Registry;
02. if( $HKEY_LOCAL_MACHINE-
>Create( "Software\\ActiveState\\Perl5", $Key ) )
03. {
04.     # We come here if the key was successfully created
```

```

05.    # or if the key already existed and the Create() method
06.    # successfully opened the key.
07.    #...process registry key...
08.    $Key->Close();
09. }
10. else
11. {
12.     print "Could not create the key. It may not exist.\n";
13. }

```

## **Connecting to a Registry**

Because administrators occasionally have to manipulate the Registry on remote machines, the Win32 API defines a function that will connect to a remote Registry. This is the `Connect()` method:

```
$RegistryObject->Connect( $Machine, $Root );
```

The first parameter (`$Machine`) is the name of a machine. This needs to be a proper machine name in the format of `\MachineName`.

The second parameter (`$Root`) is a scalar variable that will be set to a new Registry object. You will use this later to call other methods.

The `Connect()` method is called from one of the predefined root keys listed in [Table 3.13](#). If successful, the scalar variable passed into the second parameter will be a Registry object that represents the root key on the remote machine used to call the method. The return value, if successful, is `TRUE`; otherwise, it is `FALSE`.

Assume that you use this method from the `$HKEY_LOCAL_MACHINE` object:

```
$HKEY_LOCAL_MACHINE->Connect( "\\\ServerA", $ServerRoot );
```

The result is that you will have a new Registry object that represents the `HKEY_LOCAL_MACHINE` root key on the machine `\ServerA` that `$ServerRoot` represents. Anywhere you need to access something from the `HKEY_LOCAL_MACHINE` key on the remote machine (`\ServerA`), you would just use `$ServerRoot`. [Example 3.16](#) demonstrates this.

### **Note**

*Some versions of Win32::Registry did not expose the Connect() function, so it was up to the author to hack the REGISTRY.PM file and add one. This can be done by adding the following code to the REGISTRY.PM file:*

```
sub Connect
{
    my $self = shift @_;
    if( $#_ != 1 )
    {
```

```

    die "usage: Connect( $Machine, $KeyRef )";
}
my($Machine) = @_;
my( $Result, $SubHandle );
$Result = Win32::Registry::RegConnectRegistry( $Machine, $self-
>{handle},
$SubHandle );
$_[1] = _new( $SubHandle );
if( !$_[1] )
{
    return( 0 );
}
return( $Result );
}

```

### **Example 3.16 Processing Registry entries on all BDCs in a domain**

```

01.  use Win32::Registry;
02.  use Win32::NetAdmin;
03.  my $Domain = "Accounting";
04.  my @BDCs;
05.  Win32::NetAdmin::GetServers( '', $Domain,
SV_TYPE_DOMAIN_BAKCTRL, \@BDCs );
06.  foreach my $Machine ( @BDCs )
07.  {
08.      my $Root;
09.      if( $HKEY_LOCAL_MACHINE->Connect( $Machine, $Root ) )
10.      {
11.          print "Successfully connected to $Machine\n";
12.          my $Key;
13.          # We have a connection to the BDC...
14.          if( $Root->Open( "Software\\ActiveState", $Key ) )
15.          {
16.              #...process registry key...
17.              $Key->Close();
18.          }
19.          $Root->Close();
20.      }
21.      else
22.      {
23.          print "Unable to connect to $Machine's registry\n";
24.      }
25.  }

```

### **Managing Registry Keys**

Registry keys can contain values and subkeys. The values have data associated with them. The names of these elements can be a bit confusing because their relatives in INI files use similar names but have different meanings. In an INI file, for example, a key is the equivalent of a Registry value.

Likewise, an INI file has keys associated with values, but the Registry has values associated with data. This seems pretty screwy, but it is not too difficult to get used to.

## Tip

*The Registry functions are not case sensitive with respect to key and value names. When specifying either a key name or a value name, you can mix case if you like. To open a key, for example, the following two statements are considered valid and the same:*

```
$HKEY_CURRENT_USER->Open( "Software\\Microsoft", $Key);
$HKEY_CURRENT_USER->Open( "SOFwarE\\mICROsoft", $Key);
```

Any `Win32::Registry` object can be used to open a key, not just the predefined root keys. This means that a `Win32::Registry` object created by successful calls to the `Open()` or `Create()` methods can be used to open subkeys. In [Example 3.17](#), the

"`HKEY_LOCAL_MACHINE\Software\ActiveState\Perl5`" key is opened twice. The first time it is opened as a subkey from the `$Key` object, and the second time it's opened by using a root key object. This illustrates that it really does not matter how you open the keys; either method will work.

### **Example 3.17 Opening subkeys**

```
01. use Win32::Registry;
02. my $Key;
03. if( $HKEY_LOCAL_MACHINE->Open( "Software\\ActiveState",
$Key ) )
04. {
05.     my $SubKey;
06.     # Open a subkey using a previously created registry object
07.     if( $Key->Open( "Perl5", $SubKey ) )
08.     {
09.         #...process registry subkey...
10.         $SubKey->Close();
11.     }
12.     # Open the same subkey using a root key object
13.     if( $HKEY_LOCAL_MACHINE-
>Open( "Software\\ActiveState\\Perl5",
14.                     $SubKey ) )
15.     {
16.         #...process registry subkey...
17.         $SubKey->Close();
18.     }
19.     $Key->Close();
20. }
21. else
22. {
23.     print "Could not open the key. It may not exist.\n";
24. }
```

Generally, if you are processing keys by recursively calling a function, it is easier to pass a new Registry object into the recursive function, as in [Example 3.18](#).

### **Example 3.18 Processing keys using recursive functions**

```
01. use Win32::Registry;
02. my $Key;
03. my $Path = "Software\\ActiveState";
04. if( $HKEY_LOCAL_MACHINE->Open( $Path, $Key ) )
05. {
06.     # Dump all value names and sub keys...
07.     MyProcessKey( $Path, $Key );
08.     $Key->Close();
09. }
10.
11. sub MyProcessKey
12. {
13.     my( $Path, $Key ) = @_;
14.     my @KeyList;
15.     my %ValueList;
16.
17.     # Dump the key's values...
18.     print "$Path\n";
19.     $Key->GetValues( \%ValueList );
20.     foreach my $Value ( sort( keys( %ValueList ) ) )
21.     {
22.         print "\t$value : $ValueList{$Value}[2]\n";
23.     }
24.     # Process all subkeys
25.     $Key->GetKeys( \@KeyList );
26.     foreach my $SubKeyName ( sort( @KeyList ) )
27.     {
28.         my $SubKey;
29.         if( $Key->Open( $SubKeyName, $SubKey ) )
30.         {
31.             MyProcessKey( "$Path\\$SubKeyName", $SubKey );
32.             $SubKey->Close();
33.             print "\n";
34.         }
35.     }
36. }
```

## **Querying Registry Keys**

You can retrieve information about a key, such as how many subkeys and how many values it contains, by using the `QueryKey()` method:

```
$RegistryObject->QueryKey( $KeyClass, $NumOfKeys, $NumOfValues );
```

The first parameter (`$KeyClass`) is a scalar variable that will be set to the class of the key. This class is vaguely defined in the Microsoft SDK, but it seems to be inherited from previous versions

of Windows when a Registry key could have only one value (or more accurately, one data item associated with the key). In the Win32 world, this is equivalent to the default value for a key—the data that is assigned to a value with no name. (The name is an empty string.)

The second parameter (`$NumOfKeys`) is a scalar variable that will be set to the number of subkeys contained in the key that the `$RegistryObject` represents.

The third parameter (`$NumOfValues`) is a scalar variable that will be set to the number of values that the key has.

The `QueryKey()` method is handy to determine how many subkeys and values a key has; other than that, however, it is really not of much value, as illustrated in [Example 3.19](#).

If the `QueryKey()` method is successful, it returns `true`; otherwise, it returns `false`.

### ***Example 3.19 Querying a Registry key***

```
01. use Win32::Registry;
02. my $Path = 'Software\Microsoft\Internet Explorer';
03. my $Key;
04. if( $HKEY_CURRENT_USER->Open( $Path, $Key ) )
05. {
06.     my( $MyClass, $NumOfKeys, $NumOfValues );
07.     if( $Key->QueryKey( $MyClass, $NumOfKeys, $NumOfValues ) )
08.     {
09.         printf( "The '%s' key contains %d subkeys and %d
values.\n",
10.                 $Path,
11.                 $NumOfKeys,
12.                 $NumOfValues );
13.     }
14.     $Key->Close();
15. }
```

### ***Retrieving Subkey Names***

For the sake of enumerating, you can retrieve a list of subkeys that are in a particular key. This can be done with the `GetKeys()` method:

```
$RegistryObject->GetKeys( \@SubKeys );
```

The only parameter is a reference to an array that will be populated with the names of each subkey.

If the method is successful, it will return a `true` value, and the array will be filled with subkey names. Otherwise, the method returns `false`. [Example 3.18](#) makes use of this method.

### ***Removing Registry Keys***

A key can be removed, or deleted, from the Registry. Deleting a key is done using the `DeleteKey()` method:

```
$RegistryObject->DeleteKey( $KeyName );
```

The only parameter passed in (`$KeyName`) is the name of the key. This is a path to a name relative to the `$RegistryObject`.

If the key was successfully deleted, `DeleteKey()` returns `TRUE`; otherwise, it returns `FALSE`.

[Example 3.20](#) illustrates the `DeleteKey()` method.

## Warning

*When deleting keys on a Windows 95/98/ME machine, care should be used because you can easily delete a key with several subkeys. Consider that if you delete the HKEY\_CURRENT\_USER\Software key, all software configuration for the user will be lost. Likewise, deleting HKEY\_LOCAL\_MACHINE\System will remove the configuration for Windows; all system drivers and such will be lost.*

*On Windows NT/2000/XP platforms, the `DeleteKey()` method will not delete a key if it contains subkeys. A script will have to recursively enumerate each key and remove all its subkeys before deleting its parent key.*

## Warning

*Opening a key does not prevent it from being deleted. If you open the HKEY\_LOCAL\_MACHINE\Software\Microsoft key, for example, you can still delete the HKEY\_LOCAL\_MACHINE\Software key (which will remove all subkeys, including the Microsoft key you opened).*

### **Example 3.20 Deleting a Registry key**

```
01. use Win32::Registry;
02. my $Path = "Software\\ActiveState\\Perl5";
03. if( $HKEY_LOCAL_MACHINE->DeleteKey( $Path ) )
04. {
05.     print "Successfully deleted $Path.\n";
06. }
07. else
08. {
09.     print "Error deleting $Path key: $^E.\n";
10. }
```

### **Retrieving Registry Key Value Data**

Before getting too far into a discussion on how to fetch values from the Registry, it is worth noting that two methods can do this: `QueryValue()` and `QueryValueEx()`. The former exists for legacy reasons. The latter is the method you should use. It is discussed later in this section.

Any key can have any number of values. Each value has data associated with it. This data can be number, binary, or text data. To retrieve the data that a value is associated with, you can use the `QueryValue()` method:

```
$RegistryObject->QueryValue( $SubKeyName, $Value );
```

The first parameter (`$SubKeyName`) is the name of a subkey that will be queried.

The second parameter (`$Value`) will be set with the data associated with the default value for the specified key.

Chances are that `QueryValue()` is not what you think it is. If you want to query a value in a key, this will not do it for you. Let me explain.

This method calls the Win32 API's `RegGetValue()` function, which was introduced in Windows 3.1. Back in Windows 3.1, there was no concept of a key having multiple values. Actually, there was no difference between a value and a key. This function exists only for backward compatibility with Windows 3.1. Because Win32 Perl has never existed in the Windows 3.1 environment, however, there is some question as to why the `QueryValue()` function was ever exposed by the `Win32::Registry` extension.

The `QueryValue()` method retrieves the default data value from the key specified. This is of little value to most programmers because they normally need to retrieve data for a named value (as opposed to the unnamed values).

The `QueryValueEx()` method enables you to retrieve data for a value name of " " (an empty string). This is the same as using the `QueryValue()` method. Because of this, you should consider `QueryValue()` to be obsolete and of little use. Instead, use `QueryValueEx()`.

A much better way of querying a value's data is using the `QueryValueEx()` method:

```
$RegistryObject->QueryValueEx( $ValueName, $DataType, $Data );
```

The first parameter (`$ValueName`) is the name of the value to be retrieved. This can be an empty string.

The second parameter (`$DataType`) is a scalar variable that will be set to contain the data type.

[Table 3.14](#) lists the data types.

The third parameter (`$Data`) is a scalar variable that will be set to contain the data of the value.

If the `QueryValueEx()` method call is successful, it will return a `TRUE` value, and the second and third parameters' variables are set to contain their respective information. Otherwise, the method returns `FALSE`.

In some builds of both ActiveState and core distribution versions of this extension, `QueryValueEx()` was not exported by the extension's module. If you run into this, you can call the extension's function directly by using the following:

```
Win32::Registry::RegQueryValueEx( $RegistryObject->{handle},  
$ValueName,  
$Reserved, $DataType, $Data );
```

Notice that this is not called as an object's method but instead as an extension's function. The first parameter is the Registry's handle from the Registry object. This is just the "handle" key from the Registry object.

The second parameter (`$ValueName`) is the name of the value.

The third parameter (`$Reserved`) is a reserved setting that directly translates to a reserved setting in the Win32 API function. Always specify a zero (`0`) or `undef`.

The fourth parameter (`$DataType`) is a scalar variable that will be set to contain the data type. [Table 3.14](#) lists the data types.

The fifth parameter (`$Data`) is a scalar variable that will be set to contain the data of the value.

Just like the method version of this function, if `RegQueryValueEx()` is successful, it will return a `TRUE` value, and the second and third parameters' variables are set to contain their respective information. Otherwise, the method returns `FALSE`.

## **Setting Registry Key Values**

To create a value in a key, you use the `SetValue()` method:

```
$RegistryObject->SetValue( $KeyName, $Type, $Data );
```

The first parameter (`$KeyName`) is the name of a key that will be created.

The second parameter (`$Type`) is the type of data that the value will hold and must be `REG_SZ` (see [Table 3.14](#)).

The third parameter (`$Data`) is the data to be associated with the value name.

The `SetValue()` method will create a subkey with the name specified by the first parameter. This new key will contain one value that will have no name, and its data will be set to the data passed in as the third parameter. If successful, `SetValueEx()` returns a `TRUE` value; otherwise, it returns `FALSE`.

The `SetValue()` method reflects the same problem as the `QueryValue()` method. In most cases, a programmer will want to set a value in a key (not create a new key with a nonnamed value). You should consider `SetValue()` as obsolete and of little value. Instead use `SetValueEx()`:

```
$RegistryObject->SetValueEx( $ValueName, $Reserved, $DataType,  
$Data );
```

The first parameter (`$ValueName`) is the name of the value. If this value does not exist, it will be created.

The second parameter (`$Reserved`), oddly enough, is not used. You should always pass a zero (`0`) or `undef` as this parameter. The Win32 API requires this parameter. Because it is always `0`, however, I do not know why the authors of the extension decided to expose it in the Perl interface.

The third parameter (`$DataType`) is the data format that will be stored. See [Table 3.14](#) for a list of data types.

The fourth parameter (`$Data`) is the actual data to be stored.

If a call to the `SetValueEx()` method is successful, the key's specified value will be set with the specified data. A `true` value will also be returned. If it fails, `FALSE` is returned. The code in [Example 3.21](#) shows how to use the `SetValueEx()` method.

### ***Example 3.21 Modifying a value's data***

```
01. use Win32::Registry;
02. my $Path = "Software\Microsoft\Office\9.0\Word\Options";
03. my $ValueName = "AutoSave-Path";
04. my $Key;
05. if( $HKEY_CURRENT_USER->Open( $Path, $Key ) )
06. {
07.     my( $DataType, $Data );
08.     if( $Key->QueryValueEx( $ValueName, $DataType, $Data ) )
09.     {
10.         if( ( REG_SZ == $DataType ) || ( REG_EXPAND_SZ ==
$DataType ) )
11.         {
12.             # If we were autosaving to the C: drive change it
13.             # to the D: drive.
14.             $Data =~ s/^c:/d:/i;
15.             $Key->SetValueEx( $ValueName, 0, $DataType, $Data );
16.         }
17.     }
18. }
```

***Table 3.14. Registry Data Types***

<b>Registry Data</b>	<b>Type Description</b>
REG_SZ	A typical character.
REG_EXPAND_SZ	A character string that has embedded environmental variables that need to be expanded.
REG_MULTI_SZ	An array of null-terminated strings. The end of the list is terminated by a null character (so there should be two null characters at the very end of the list).
REG_BINARY	Any form of binary data. This could be of any length.

**Table 3.14. Registry Data Types**

Registry Data	Type Description
REG_DWORD	A 32-bit number.
REG_DWORD_LITTLE_ENDIAN	A 32-bit number in little endian format (where the most significant byte of the word is the high-order word). Typically, this is the same as REG_DWORD. This type is used on Intel processors.
REG_DWORD_BIG_ENDIAN	A 32-bit number in big endian format (where the most significant byte of the word is the low-order word). Motorola processors make use of this format.
REG_LINK	A symbolic link in the Registry. This is the data type used to map root keys such as HKEY_CURRENT_USER to another key within the Registry.
REG_NONE	No defined value.
REG_RESOURCE_LIST	A device driver resource list.

### ***Listing Registry Key Values***

To get a list of values from an opened key, you can use the `GetValues()` method:

```
$RegistryObject->GetValues( \%ValueList );
```

The only parameter passed in this method is a reference to a hash. If the method is successful, it will return a `TRUE` value and populate the hash; otherwise, it returns `FALSE`.

When the `GetValues()` method completes successfully, the hash is populated with keys that represent the names of the values in the key. Each of the hash's keys is associated with an array consisting of the three elements listed in [Table 3.15](#). Notice that element 0 of the array is identical to the hash's key name. [Example 3.18](#) demonstrates this method.

**Table 3.15. The Array Returned by `GetValues()`**

Array Element	Description
Element 0	The name of the value.
Element 1	The data type of the value's data. Refer to the list of data types in <a href="#">Table 3.14</a> .
Element 2	The value's data.

### ***Removing Registry Key Values***

You can delete a value just as you can delete a key. This is done by using the `DeleteValue()` method:

```
$RegistryObject->DeleteValue( $ValueName );
```

The only parameter specified in this method is the name of the value to be deleted.

If `DeleteValue()` is successful, it will return `TRUE`; otherwise, it returns a `FALSE` value.

### **Closing an Open Registry Key**

After you have finished with a key you have opened, you need to close it using the `Close()` method:

```
$RegistryObject->Close();
```

If successful, `close()` returns `TRUE` and the Registry object no longer exists. Any further method calls using this object will fail. If the method fails, it returns a `FALSE` value.

### **Managing Registry Hives**

There are many reasons why you might need to save a key (and its values and subkeys) to a file. For example, you might want to create a backup copy or identical keys on multiple machines. This file is called a *hive*. These hives can be loaded and unloaded from the Registry whenever needed. As another example, a user's Registry profile is stored in a hive. When the user logs on, that hive is loaded and mapped to the `HKEY_CURRENT_USER` root.

#### **Saving a Hive**

Saving a Registry hive is an easy way to migrate Registry modifications across several machines. By saving a hive to a file, a script can later either load the hive or replace an existing key with the saved hive. This way, you can configure a program, save the hive that contains the program's configuration values to a file, and then use that file on other Registries. Descriptions of how to load hives and replace keys with hives follow this section.

To save a hive (a key with all its values and subkeys), you use the `Save()` method:

```
$RegistryObject->Save( $File );
```

The only parameter passed in `Save()` is the name of a file that will be created.

If successful, the `Save()` method will save the key and all its values and subkeys to a binary file that will be marked as a hidden, read-only system file. If the file already exists, the method will fail.

The resulting file is considered to be a hive. [Example 3.22](#) demonstrates saving a Registry hive.

#### **Example 3.22 Saving a Registry hive**

```
01. use Win32::Registry;
02. my $Path = "Software\\ODBC";
03. my $FilePath = "c:\\temp\\ODBC.key";
04. my $Key;
05. if( $HKEY_LOCAL_MACHINE->Open( $Path, $Key ) )
```

```

06. {
07.     if( $Key->Save( $FilePath ) )
08.     {
09.         print "Successfully saved the $Path hive to $FilePath.\n";
10.    }
11. else
12. {
13.     print "Failed to save the hive: $^E.\n";
14. }
15. $Key->Close();
16. }
17. else
18. {
19.     print "Error opening Registry key: $^E.\n";
20. }

```

## Note

*It is very important to note that all methods and functions in the `Win32::Registry` extension that make use of files (such as `Save()` and `Load()`) assume that the file's path is relative to the machine housing the Registry. This means that if you are connected to a remote Registry and load a hive from `c:\temp\hive.reg`, the file must be on the remote machine's `c:\temp` directory.*

*Additionally, the Microsoft Win32 documentation specifies that Registry files created with any file extension on a FAT-formatted drive will fail to load. This does not seem to be the case in Windows 95, but your mileage may vary.*

## Loading a Hive

After a hive has been saved to a file, it can be loaded into a Registry where it can be edited. This is what the `Load()` method is all about:

```
$RegistryObject->Load( $KeyName, $File );
```

The first parameter (`$KeyName`) is the name of the key that will represent the hive. This name must not already exist. You cannot load a hive over an existing key; if you try, the method will fail.

The second parameter (`$File`) is the path to a saved hive file.

The `$RegistryObject` that calls this method must be one of either `$HKEY_LOCAL_MACHINE` or `$HKEY_USERS`. It cannot be any nonroot key you have opened.

## Note

*The `Load()` method can only be called from a true root key such as `HKEY_LOCAL_MACHINE` and `HKEY_USERS`. Other root keys such as `HKEY_CURRENT_USER` are not truly root keys because they are just links to other nonroot keys.*

*For example, the root `HKEY_CURRENT_CONFIG` is really just a link to `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Hardware Profiles\Current`. Because `HKEY_CURRENT_CONFIG` is not a true root key, the method cannot be called from it.*

If the `Load()` method is successful, the hive will have been loaded and can be accessed by opening the key name specified as the first parameter to this method. The `Load()` method will return `TRUE` if successful; otherwise, it returns `FALSE`. See [Example 3.23](#) to see the `Load()` method in action.

## Warning

*Registry key names can consist of any character except the backslash () because the backslash is used to delimit keys in a Registry path. If you try to create a key containing a backslash in the Registry Editor, it will fail and give you an error message. The Win32::Registry extension, however, enables you to create key names with a backslash.*

## Tip

*It is in your best interest to not create keys with backslashes because you might not be able to open the key to either modify or remove it.*

## Unloading a Hive

If you have a recent version of the `Win32::Registry` extension that has a method for unloading a hive, you can use the `UnLoad()` method:

```
$RegistryObject->UnLoad( $KeyName );
```

The only parameter is the name of the key with which you had previously loaded a hive.

This function and method will return `TRUE` if they successfully unloaded the hive; otherwise, `FALSE` is returned.

When the hive is unloaded, it is saved so that any changes made to the hive (any keys that have been renamed, deleted, or added, or any values or data changed) will be saved to the hive's file. See [Example 3.23](#) to see how to unload a Registry hive.

You can only unload hives from one of the root keys. Specifying the handle of any key other than a root key (or calling the method from any object other than a root object) will result in the function failing.

The `UnLoad()` method is not exported in all versions of the `Win32::Registry` extension. Hopefully, this will change in future versions of the module. In the meantime, you can unload any hive by using the `RegUnloadKey()` function:

```
Win32::Registry::RegUnLoadKey( $Handle, $KeyName );
```

The first parameter (`$Handle`) is the handle to a root Registry key such as `HKEY_LOCAL_MACHINE`. The handle is a numeric value held in the Registry object's "handle" member. You can access this as `$HKEY_LOCAL_MACHINE->{handle}`, assuming that you are looking for the local machine's root handle.

The second parameter (`$KeyName`) is the name of the key with which you had previously loaded a hive.

### ***Example 3.23 Loading and unloading Registry hives***

```
01. use Win32::Registry;
02. my $HiveFilePath = "c:\\temp\\ODBC.key";
03. my $KeyName = "My temporary ODBC hive";
04. my $Key;
05.
06. # Load the hive into this registry.
07. $HKEY_LOCAL_MACHINE->Load( $KeyName, $HiveFilePath )
08.     || die "Can not load the hive";
09. if( $HKEY_LOCAL_MACHINE->Open( $KeyName, $Key ) )
10. {
11.     #...process key...
12.     $Key->Close();
13. }
14. # Remove the hive from this registry (saving the contents).
15. $HKEY_LOCAL_MACHINE->UnLoad( $KeyName );
```

### ***Restoring Registry Keys***

Two functions in the `Win32::Registry` extension are wonderfully valuable but are just not exposed by the extension's module. Because of this, you must access them as functions. Just like the other such functions, the extension might change in future releases to provide methods to access these functions.

The first of these functions, `RegRestoreKey()`, restores a key with another key that was saved to a file (refer to the "Saving a Hive" section earlier in this chapter to learn how to save a key to a file). This function takes the Registry key values and subkeys from a given file and overwrites any currently existing keys. This is great if you want to make backups of important Registry keys and later restore them. The syntax for the `RegRestoreKey()` function is as follows:

```
Win32::Registry::RegRestoreKey( $Handle, $File[, $Flag ] );
```

The first parameter (`$Handle`) is a handle to an opened key. You can use any key that has been opened (by using either the `Open()` or `Create()` method). Access the handle by referring to the "handle" member of the key's object, as in:

```
$Key->{handle}
```

The second parameter (`$File`) is the name of a file that has had a previously saved key.

The third parameter (`$Flag`) is optional and specifies the volatility flag. This flag can either be `0` or `REG_WHOLE_HIVE_VOLATILE`. If `REG_WHOLE_HIVE_VOLATILE` is specified, the hive will be active until the next time the machine is rebooted. After reboot, the key will resort back to its original state. This flag can be specified only on keys that exist under `HKEY_USERS` or `HKEY_LOCAL_MACHINE` roots. The default for this parameter is `0`.

If successful, the key will be replaced by the contents of the hive file, and the function will return `TRUE`. `FALSE` is returned on failure.

If a key is restored, all its original values and subkeys are lost and are replaced with the contents of the specified file.

The other function that restores a key but also provides a backup for the previous key is the `RegReplaceKey()` function:

```
Win32::Registry::RegReplaceKey( $Handle, $KeyName, $File,
$BackupFile )
```

The first parameter (`$Handle`) is a handle to an opened key. You can use any key that has been opened (by using either the `Open()` or `Create()` method). Access the handle by referring to the "handle" member of the key's object, as in:

```
$Key->{handle}
```

The second parameter (`$KeyName`) is the name of the key to be replaced.

The third parameter (`$File`) is the name of a file that has had a previously saved key. The contents of this file will replace the specified key.

The fourth parameter (`$BackupFile`) is the name of a file that will be created. A backup of the key will be stored into this file before it is replaced.

If the `RegReplaceKey()` function is successful, it will return `TRUE`, a backup file will have been created containing the original key (its values and subkeys), and the key will have been replaced by the contents of the file specified in the third parameter. If the `RegReplaceKey()` function fails, it will return `FALSE`.

## Managing the Event Log

The Windows NT operating system contains a mechanism to track events and messages generated by the operating system, applications, services, and drivers. This is similar to the UNIX SYSLOG daemon. The mechanism is known as the Event Log, and it can be a powerful tool (if not the only tool) to capture important messages that can predict network failures, nonworking devices, and other such information.

Win32 Perl can access the database of messages that makes up the Event Log by using the `Win32::EventLog` extension.

## ***Opening the Event Log***

Before performing any actions on the Event Log, you need to open the Event Log. The function used to open the log will result in a new `Win32::EventLog` object that will be used to perform all other methods.

To open the Event Log, you can use the `Open()` method:

```
Win32::EventLog::Open( $Object, $Source [, $Machine ] );
```

The first parameter (`$Object`) is a scalar variable that will be set to the newly created Event Log object. This object will be used to call other methods.

The second parameter (`$Source`) is the source name. This is a string that will be used to identify the origin of any events that might be stored in the log. This is usually some string that is meaningful to whichever user will be looking at the logs. By default, there are three sources on a Win32 machine: `Application`, `Security`, and `System`. A script could, however, create its very own source such as `Perl Database Manager` or `Joel's funky event source`. This could be an empty string.

The third parameter (`$Machine`) is optional and represents the computer whose Event Log this object will be connected to. Any methods that the created `Win32::EventLog` object performs will be done on this machine's Event Log. The default for this is the local machine. This is just the computer name (not the proper name with prepended backslashes).

If successful, `TRUE` is returned, and a new `Win32::EventLog` object is created representing the Event Log on the specified machine. If this fails, `Open()` returns a `FALSE` value.

In newer builds of the `Win32::EventLog` extension, another function exists for creating the object. That function is the Perl new command:

```
$Object = new Win32::EventLog( $Source[, $Machine ] );
```

All the parameters are the same as in the `Open()` function except that the `$Object` scalar is moved out of the parameter list.

## ***Counting Events***

Because the Event Log is a database of messages, you can query the database to determine what the latest message number is by using the `GetNumber()` method:

```
$EventlogObject->GetNumber( $Number );
```

The only parameter is a scalar variable that will be set to the number of records (or events) in the Event Log database.

If the `GetNumber()` method is successful, it returns `TRUE`, and the scalar variable passed into the method is set to the number of records in the database. The method returns `FALSE` if it fails.

Similar to the `GetNumber()` method is the `GetOldest()` method, which will return the record number of the oldest event in the database:

```
$EventlogObject->GetOldest( $Record );
```

The only parameter passed in is a scalar variable that will be set to the record number that is the oldest record in the Event Log database.

If this is successful, it returns `TRUE` and sets the scalar variable to the record number of the oldest record in the database. Failure is indicated by a `FALSE` return value. Together these methods are used to correctly compute the offset required by the `Read()` method that follows this section.

## ***Reading Event Messages***

Reading messages from the Event Log is fairly straightforward. You just use the `Read()` method:

```
$EventlogObject->Read( $Flags, $Record, $Event );
```

The first parameter (`$Flags`) refers to the flags that indicate how the read process is handled. This can be a combination of values from [Table 3.18](#) logically OR'ed together.

The second parameter (`$Record`) is the record number from the database that you want to read. This number is ignored unless the `EVENTLOG_SEEK_READ` flag is specified in the first parameter.

The third parameter (`$Event`) is a reference to a hash. This should be in the form of a scalar (`$Event`) as opposed to a hash (`\%Event`) reference because some older versions of this extension did not correctly work with hash references.

If the method is successful, the hash is filled with the structure defined in [Table 3.16](#), and a `TRUE` value is returned. Otherwise, `FALSE` is returned. [Example 3.24](#) shows how the `Read()` method is used.

### ***Example 3.24 Printing all system errors over the past week***

```
01. use Win32::EventLog;
02. # Enable message retrieval...
03. $Win32::EventLog::GetMessageText = 1;
04. my $SecPerWeek = 7 * 24 * 60 * 60;
05. my $Now = time();
06. my $WeekAgo = $Now - $SecPerWeek;
07. my $Num;
08. my $Event = new Win32::EventLog( "System", "" )
09.           || die "Unable to open event log.\n";
10. $~ = "EVENT_MESSAGE";
11. if( $Event->GetNumber( $Num ) )
12. {
```

**Table 3.16. The Event Log Record Hash**

Hash Key	Description
----------	-------------

**Table 3.16. The Event Log Record Hash**

Hash Key	Description
Source	The source of the event message. This is normally the name of the application or device that generates the event. This name is mapped to an application or DLL file that contains information on how to format the event message.
Computer	The name of the computer that generated the event.
Length	The number of bytes of data associated with the event. This data is found in the <code>Data</code> key.
Data	Binary data of any length that is associated with the message. The length of this data is found in the <code>Length</code> key. Usually, this is some sort of data that is reportend when an application generates an error. For most practical purposes, this key is ignored.
Category	This represents a category of the event. This is usually a number defined by the source that generated the event. You need to know what this number represents to the particular source for it to be of any use.
RecordNumber	The record number in the Event Log database of the event.
TimeGenerated	The time at which the event was generated. This number is represented as the number of seconds since January 1, 1970. You can use this number as if it were generated from Perl's <code>time()</code> function.
Timewritten	The time at which the event was physically written to the database. This value is managed by the operating system and cannot be written by a Perl script. This number is represented as the number of seconds since January 1, 1970. You can use this number as if it were generated from Perl's <code>time()</code> function.
EventID	The ID number of the event. This number is specific to the source that generated the event. This ID number's meaning changes from source to source.
EventType	This represents the type of event. The different event types are listed in <a href="#">Table 3.17</a> .
ClosingRecordNumber	This is reserved by the Win32 API and currently has no meaning. Ignore this key.
Strings	The different data strings used to format the event's message. These strings can be passed in an anonymous array.
Message	If the <code>\$Win32::EventLog::GetMessageText</code> variable is set to <code>true (1)</code> then this key is populated with the actual Event Log message text. This does not always work on all messages.

**Table 3.17. Event Log Event Types**

Event	Type Description
-------	------------------

**Table 3.17. Event Log Event Types**

Event	Type Description
EVENTLOG_ERROR_TYPE	The event was an error.
EVENTLOG_WARNING_TYPE	The event was a warning.
EVENTLOG_INFORMATION_TYPE	The event was used to inform that some event took place.
EVENTLOG_AUDIT_SUCCESS	The event was a successful audit. This occurs if an administrator has specified that an object be audited (such as a file, directory, or Registry key), and a user successfully accessed the object.
EVENTLOG_AUDIT_FAILURE	The event was a failed audit. This occurs if an administrator has specified that an object be audited (such as a file, directory, or Registry key), and a user was unable to access the object.

**Table 3.18. Event Log Reading Flags**

Flag	Description
EVENTLOG_FORWARDS_READ	The events are read from the database forward in chronological order.
EVENTLOG_BACKWARDS_READ	The events are read from the database backward in chronological order.
EVENTLOG_SEEK_READ	Reads the specified record number.
EVENTLOG_SEQUENTIAL_READ	Reads the next record in the database.

### **Writing to the Event Log**

You can create your own entries in the Event Log. This can be used when automated scripts run and errors occur. This is done with the `Report()` method:

```
$EventlogObject->Report( \%Event );
```

The only parameter is a reference to an event hash, as described in [Table 3.11](#).

If the method is successful, an entry is added to the Event Log database consisting of the data contained in the hash whose reference was passed into the method. A `true` value is also returned. A `false` value is returned if the method fails.

### **Clearing the Event Log**

The Event Log database can become quite large. Because of this, you might want to clear the database using the `Clear()` method:

```
$EventlogObject->Clear( $File );
```

The only parameter passed in is a filename.

Before the Event Log is cleared, it will be backed up into the file specified. If this is successful, the `Clear()` method will return `TRUE`; otherwise, it returns `FALSE`.

## Note

*When you clear the Event Log, it is automatically closed. Because of this, the Event Log object that called the `Clear()` method no longer is valid and should be discarded.*

## **Backing Up the Event Log**

Just as the `Clear()` method will create a backup log file of the Event Log, so does the `BackupEventLog()` function. The only difference is that it does not empty the Event Log. This function is so useful that I do not understand why it was not exposed to the Event Log object by means of a method. Instead, you must access it directly by means of a function call:

```
Win32::EventLog::BackupEventLog( $Handle, $File );
```

The first parameter (`$Handle`) is a handle to an open Event Log. You can specify the "handle" member of the Event Log object for this parameter.

The second parameter (`$File`) is the name of the backup file that will be created.

If the `BackupEventLog()` function is successful, the Event Log has been backed up to the file, and the function returns `TRUE`. `FALSE` is returned to indicate a failure.

## **Closing the Event Log**

It is good practice to always close any opened Event Log files using the `CloseEventLog()` function:

```
$EventLogObject->Close();
```

For some very odd reason, in the early versions of the `Win32::EventLog` extension, the authors failed to expose the method that closes an opened database. If you are using an old version that complains when you call the `Close()` method, try using the following:

```
Win32::EventLog::CloseEventLog( $Handle );
```

The only parameter accepted is a handle to an open Event Log. You can specify the "handle" member of the Event log object for this parameter.

If the Event Log is successfully closed, it returns `TRUE`; otherwise, it returns `FALSE`. Refer to the code in [Example 3.24](#) to see how the `CloseEventLog()` function is used.

## Summary

Administrators using the Win32 extensions should consider themselves lucky to have such a war chest of valuable functions to assist them in their daily chores of running a network of users and machines.

The Win32 extensions provide a wide range of functionality to manage accounts. Not only user but also machine accounts can be managed. This is not limited to the general adding and removing of accounts; it also includes renaming, disabling, and changing passwords as well as modifying the current state of an account.

As if account management weren't enough, the Win32 extensions also provide access to group management. Creating, modifying, and removing both local and global groups are supported. With these functions, users can be added to, enumerated, and removed from groups. Additionally, you can also check a user's membership in a group.

User accounts and groups are not the only things that need management. Administrators need to also manage machines. The Win32 extensions provide access to the Event Logs to both query events and add them. Win32 Perl extensions also contain a set of functions that provides access to the Registry, enabling administrators to modify or restore components.

## Chapter 4. File Management

Win32 does a remarkable job of providing access to data. It provides copious amounts of information regarding its data sources, such as disk files. These files are incredibly important for storing user data as well as for supporting the operating system. This chapter focuses on managing files and directories on a machine. It covers file attributes, advanced file information, permissions, shortcuts, and monitoring directories for changes.

Files contain not only data, but also meta information that describes the file itself (such as whether it has been altered, whether it is a system file, its author, and its version number). This meta information, or *attributes*, is important for an administrator to track (for example, version numbers of executable and DLL files). This chapter explains how to access and manage this information.

Shortcuts are convenient to use but tedious to manage. If a directory name changes or data is moved from one server to another, many shortcuts might need to be updated. Doing this manually on all machines on a network can take quite a bit of time and patience. Descriptions of how to create and modify these shortcuts are covered later in this chapter.

Files and directories on NTFS partitions can have permissions placed on them. This is what grants or denies a user the ability to access files. For servers in a secure environment, managing this is not only important but necessary, especially when automation scripts are written that must create a directory tree and apply permissions to it. These issues are covered in detail.

Finally, this chapter covers the concept of directory monitoring. Watching a directory for changes that occur is useful for daemon services that need to know when a file size or directory name has changed.

# File Attributes

If you have ever played around with files, you know that all files and directories in the Win32 world have the four basic common file attributes listed in [Table 4.1](#) (inherited from the DOS world).

**Table 4.1. Common File Attributes**

Attribute	Description
ARCHIVE	The <code>ARCHIVE</code> attribute is set any time the file's contents change. Backup programs that back up only files with the <code>ARCHIVE</code> flag set use this.
HIDDEN	A file with the <code>HIDDEN</code> attribute set does not appear in directory listings.
READONLY	The operating system does not allow files with the <code>READONLY</code> flag set to change.
SYSTEM	The <code>SYSTEM</code> attribute specifies that a file is a system file. System files require special handling by the operating system. The main two files that make up the core of DOS, for example, are IO.SYS and MSDOS.SYS (presuming that we are talking about Microsoft's DOS, that is). These files have the system flags set so that DOS knows to handle these important files with special care. Typically, system files are rendered as invisible, so they do not appear in normal directory listings.

## What Are the File Attributes?

Win32 Perl has a standard extension known as `Win32::File` that allows for only two functions: retrieving and setting file attributes.

These two functions are fast and cover more attributes than the DOS box's `attrib` command. Whereas the `attrib` command recognizes the attributes in [Table 4.1](#), the `Win32::File` extension recognizes the attributes listed [Table 4.1](#) as well as those listed in [Table 4.2](#).

**Table 4.2. Additional File Attributes Recognized by the `Win32::File` Extension**

Attribute	Description
COMPRESSED	The file has been compressed by the operating system to conserve disk space. This does not apply to files that have been compressed by means other than the operating system (such as archiving programs like zip.exe).
DIRECTORY	This attribute distinguishes a file from a directory. It will only be set if the path is a directory.
NORMAL	When this attribute is set, all common attributes are cleared ( <code>ARCHIVE</code> , <code>HIDDEN</code> , <code>READONLY</code> , and <code>SYSTEM</code> ).

Two other attributes are of equal importance; for some reason, however, both are undocumented, and their constant names are not recognized by the extension. You can still test and set them, but

you must use the value of the constant as opposed to the constant's name. [Table 4.3](#) lists these values.

**Table 4.3. File Attributes Recognized Only by Value by the Win32::File Extension**

Attribute	Hexadecimal	Value Description
OFFLINE	0x1000	The file's data is currently not available. This can occur when the operating system has moved the data to offline storage (as when a floppy disk has been removed).
TEMPORARY	0x0100	The file was created as a temporary file and is most likely going to be destroyed by the application that created it.

Now that you know what attributes are available, it is time to see what limitations there are when using these attributes.

Not all attributes can be set. The `DIRECTORY` and `COMPRESSED` attributes are read-only. They cannot be set by the `Win32::File` extension. You would have to run the command line utility `compact` command or use the Explorer program to compress a file or directory. This will set the `COMPRESSED` bit. Using the same means can clear the bit as well. Refer to the `compact` command for directions.

The `DIRECTORY` bit is set only on directories. It would not make sense to set the `DIRECTORY` bit on a file. After all, a file is not a directory no matter what you do to it (unless, of course, you delete the file and create a directory by the same name).

You need to be aware that the `NORMAL` attribute is the most unique of the attributes. When you set the `NORMAL` attribute, all the attributes are removed (except `DIRECTORY` and `COMPRESSED`). You can use this to reset the state of a file or directory.

## Retrieving File Attributes

When you need to check the attributes on a file or directory, you should use the `GetAttributes()` function:

```
Win32::File::GetAttributes( $Path, $Attributes );
```

The first parameter (`$Path`) is the path to the file or directory for which you want the attributes. This can be a full path, a relative path, or a UNC. If the path is a directory, it does not matter if it ends with a backslash or not.

The second parameter (`$Attributes`) will receive, if successful, the attributes in a bitmap format. To test for particular attributes, you can logically AND the resulting value of this parameter with one of the constants, as in line 6 of [Example 4.1](#). If the `GetAttributes()` function is successful, it returns a 1; otherwise, it returns nothing.

### Example 4.1 Checking for the `READONLY` attribute

```
01. use Win32::File;
```

```

02. my $File = shift @ARGV || $0;
03. my $Attrib;
04. if( Win32::File::GetAttributes( $File, $Attrib ) )
05. {
06.   if( $Attrib & READONLY )
07.   {
08.     print "$File is read only.\n";
09.   }
10.   else
11.   {
12.     print "$File is NOT read only.\n";
13.   }
14. }
15. else
16. {
17.   print "Error: ", $^E = Win32::GetLastError(), "\n";
18. }

```

## Setting File Attributes

Setting the attributes of a file or a directory is rather easy when you use the `SetAttributes()` function:

```
Win32::File::SetAttributes( $Path, $Attributes );
```

The first parameter (`$Path`) is the path to a file or a directory. This can be a relative path, a full path, or a UNC.

The second parameter (`$Attributes`) is a bitmask of file attributes. You can logically OR the attribute constants together to specify a combination of attributes to set, as in line 3 of [Example 4.2](#). If the `SetAttributes()` function is successful, it returns a 1; otherwise, nothing is returned.

### **Example 4.2 Setting the *READONLY* and *HIDDEN* attributes**

```

01. use Win32::File;
02. my $File = shift @ARGV || $0;
03. my $Attrib = READONLY | HIDDEN;
04. if( Win32::File::SetAttributes( $File, $Attrib ) )
05. {
06.   print "$File is now READONLY and HIDDEN.\n";
07. }
08. else
09. {
10.   print "Error: ", $^E = Win32::GetLastError(), "\n";
11. }

```

### Note

The `DIRECTORY` and `COMPRESSED` attributes are read-only. That is, you can retrieve them with `Win32::File::GetFileAttributes()`, but you cannot set them with `Win32::File::SetAttributes()`. To set the `COMPRESSED` bit on a file or directory, you can use the `compact` command from a DOS box. You cannot set the `DIRECTORY` bit on a file, nor can you clear the bit on a directory.

## Case Study: Retrieving File Attributes with the DOS `attrib` Command

The DOS `attrib` command is used to read and set the common file attributes on files and directories. From Perl, you can shell out and run the `attrib` command using Perl's backtick, but this results in rather large memory and time penalties, as illustrated in [Example 4.3](#).

### **Example 4.3 Retrieving file attributes with the DOS `attrib` command**

```
01. my $Time = time();
02. my $iCount = 0;
03. foreach my $File ( glob( shift @ARGV || "*.*" ) )
04. {
05.     my %Attributes = GetAttributes( $File );
06.     $iCount++;
07.     print "$File has the following attributes:\n";
08.
09.     # Print each attribute for the file
10.    foreach my $Key ( sort( keys( %Attributes ) ) )
11.    {
12.        print " \u$Key" if (( $Attributes{$Key} ) );
13.    }
14.    print "\n\n";
15. }
16. my %Time = ComputeTime( time() - $Time );
17. print "\nTotal files: $iCount\n";
18. printf( "Total time: %02d:%02d:%02d\n",
19.         $Time{hour},
20.         $Time{min},
21.         $Time{sec} );
22.
23. sub GetAttributes
24. {
25.     my( $File ) = @_;
26.     my( @Output, @Attribs, %Hash );
27.     # Here we shell out using backticks.
28.     # This is a big performance hit!!
29.     @Output = `attrib $File`;
30.     (@Attribs) = ( $Output[0] =~ /^(.) (.)(.)(.)(.)/ );
31.     %Hash = (
32.         'archive' => ( $Attribs[0] eq "A" ),
33.         'system' => ( $Attribs[1] eq "S" ),
34.         'hidden' => ( $Attribs[2] eq "H" ),
35.         'read-only' => ( $Attribs[3] eq "R" ),
36.         'directory' => ( -d $File )
```

```

37.      );
38.      return( %Hash );
39.  }
40.
41. sub ComputeTime
42. {
43.     my($Temp) = @_;
44.     my(%Time);
45.     $Time{hour} = int( $Temp / 3600 );
46.     $Temp = $Temp % 3600;
47.     $Time{min} = int( $Temp / 60 );
48.     $Time{sec} = int( $Temp % 60 );
49.     return %Time;
50. }

```

As you can see by the code in [Example 4.3](#), your script will suffer due to the time it takes to shell out and process the `attrib.exe` command for each and every file processed. Thanks to the wonders of Win32 Perl, there is a much better way of doing this: the `Win32::File` extension.

To demonstrate the speed increase realized when using the `Win32::File` extension, take a look at [Example 4.4](#), which is the same script as [Example 4.3](#) with the exception of using `Win32::File::GetAttributes()` instead of shelling out to the `attrib.exe` command.

I ran both scripts on my machine, which has 152 files in the `C:\TEMP` directory. The code in [Example 4.3](#) took a total of 38 seconds to complete. The code in [Example 4.4](#) took only 9 seconds. If you look at the code carefully, you will notice that the 9 seconds includes the overhead of loading and initializing the `Win32::File` extension, not to mention that [Example 4.4](#) ran quicker while detecting more attributes.

#### **Example 4.4 Retrieving file attributes with the `Win32::File` extension**

```

01. use Win32::File;
02. my $Time = time();
03. my $iCount = 0;
04. foreach my $File ( glob( shift @ARGV || "*.*" ) )
05. {
06.     my %Attributes = GetAttributes( $File );
07.     $iCount++;
08.     print "$File has the following attributes:\n";
09.
10.    # Print each attribute for the file
11.    foreach my $Key ( sort( keys( %Attributes ) ) )
12.    {
13.        print " $Key" if (( $Attributes{$Key} ) );
14.    }
15.    print "\n\n";
16. }
17. my %Time = ComputeTime( time() - $Time );
18. print "\nTotal files: $iCount\n";
19. printf( "Total time: %02d:%02d:%02d\n",
20.           $Time{hour},

```

```

21.      $Time{min},
22.      $Time{sec} );
23.
24. sub GetAttributes
25. {
26.     my( $File ) = @_;
27.     my( $Attrib, %Hash );
28.     if( Win32::File::GetAttributes( $File, $Attrib ) )
29.     {
30.         %Hash = (
31.             'archive' => $Attrib & ARCHIVE,
32.             'system' => $Attrib & SYSTEM,
33.             'hidden' => $Attrib & HIDDEN,
34.             'read-only' => $Attrib & READONLY,
35.             'directory' => $Attrib & DIRECTORY,
36.             'compressed' => $Attrib & COMPRESSED,
37.             'temporary' => $Attrib & 0x0100,
38.             'offline' => $Attrib & 0x1000
39.         );
40.     }
41.     return( %Hash );
42. }
43.
44. sub ComputeTime
45. {
46.     my($Temp) = @_;
47.     my(%Time);
48.     $Time{hour} = int( $Temp / 3600 );
49.     $Temp = $Temp % 3600;
50.     $Time{min} = int( $Temp / 60 );
51.     $Time{sec} = int( $Temp % 60 );
52.     return %Time;
53. }

```

## File Information

In addition to the file attributes that describe the state of a file, such as `READONLY` and `HIDDEN`, the Win32 platform allows files to have additional information embedded within the file. This information ranges from the company that created the file to the original name of the file.

This information is stored as a resource within the file; therefore, it is only available on files that can have embedded file information resources, such as executable files and DLLs.

You can view this information from the Property window by right-clicking on the file in Explorer, selecting Properties, and then selecting the Version tab (if it exists). The file information will display as in [Figure 4.1](#).

**Figure 4.1. Explorer's view of a file's version information.**



Unlike the attributes discussed in the "[File Attributes](#)" section, this file information is not required; therefore, it is not found in all `.EXE` and `.DLL` files. On files that do have such information, the `Win32::AdminMisc` extension can retrieve this data.

## Retrieving File Information

To retrieve a file's information fields, you must first load the `Win32::AdminMisc` extension. After doing this, you can use the `GetFileInfo()` function:

```
Win32::AdminMisc::GetFileInfo ( $Path, \%Info );
```

The first parameter (`$Path`) is the path to a file. This file needs to be a file with an embedded file information resource, such as an executable or DLL. This path can be a local file or a UNC.

The second parameter (`\%Info`) is a reference to a hash. This will be populated with version information if it is available. [Table 4.4](#) lists the keys of the hash. It is possible that some of the hash keys might contain values of anonymous subhashes. These subhashes contain keys/value pairs that describe information related to their parent key. For example, if the `\%Info` contains a key with the name of "0x0409" (indicating the English language), it would point to an anonymous hash containing English-based info.

**Table 4.4. Information Returned by the `Win32::AdminMisc::GetFileInfo()` Function**

Hash Key	Description
CompanyName	The name of the company credited with the file's creation.

**Table 4.4. Information Returned by the Win32::AdminMisc::GetFileInfo() Function**

Hash Key	Description
FileVersion	The version of the file.
InternalName	The name used to refer to the file. The developer usually uses this.
LegalCopyright	Any copyright notice.
OriginalFilename	The original name of the file. This is handy if the file was accidentally renamed.
ProductName	The name of the product.
ProductVersion	The version of the product.
LangID	The language ID of the file in hexadecimal notation. For example, the ID 0x0409 represents U.S. English and 0x0407 represents German.
Language	The language as a string, such as "English (United States)".

If the `Win32::AdminMisc::GetFileInfo()` function is successful, it will populate the hash reference passed in as the second parameter and return a 1; otherwise, it returns a 0.

### Tip

*The `Win32::AdminMisc::GetFileInfo()` function accepts paths with both backslashes and slashes, as in [Example 4.5](#). This can be convenient when porting scripts because UNIX paths use slashes as directory delimiters.*

### Example 4.5 Using the Win32::AdminMisc::GetFileInfo() function

```

01. use Win32::AdminMisc;
02. my $Path = shift @ARGV || $^X;
03. my %Info;
04. if( Win32::AdminMisc::GetFileInfo( $Path, \%Info ) )
05. {
06.     print "File info for $Path:\n";
07.     DumpData( \%Info, "\t" );
08. }
09. else
10. {
11.     print "Error: ", $^E = Win32::AdminMisc::GetError(), "\n";
12. }
13.
14. sub DumpData
15. {
16.     my( $Info, $Indent ) = @_;
17.     foreach my $Key ( sort( keys( \%$Info ) ) )
18.     {
19.         print "$Indent $Key: ";

```

```

20.     if( "HASH" eq ref $$Info->{$Key} ) 
21.     {
22.         print "\n";
23.         DumpData( $Info->{$Key}, "$Indent\t" );
24.     }
25.     else
26.     {
27.         print $Info->{$Key};
28.     }
29.     print "\n";
30. }
31. }

```

## Shortcuts

With the introduction of Windows NT 4.0 and Windows 95, the concept of a *shortcut* was brought to the Win32 platform. A shortcut is a type of alias for a file or directory. You can use a collection of shortcuts that point to your favorite directories and files. These files could be commonly used programs or documents you read or edit often. Creating these shortcuts in a particular directory makes a convenient way to jump to your commonly used programs, files, and directories. This is what Microsoft Office products do when they create the Favorites directory.

A shortcut is like a link in the UNIX world except that it is not processed by the OS directly; instead, it is processed by an application (try accessing a shortcut using an older Windows program—it will think it is an actual file and not a link to another file or directory).

These links are handy from a graphical interface point of view. In an open dialog box, selecting a shortcut will either open the file or change to the directory that the shortcut points to.

Shortcuts are special binary files that have an `.LNK` extension, and their iconic representations on the Desktop or in Explorer have a little white box filled with a curving arrow in the lower left of the icon (see [Figure 4.2](#)). There are different types of shortcuts (such as Dial-Up Networking and printer shortcuts), but the `Win32::Shortcut` extension addresses file, program, and directory shortcuts.

**Figure 4.2. A shortcut representing Eudora Pro.**



Shortcuts consist of a collection of properties, which are documented in [Table 4.5](#).

**Table 4.5. Shortcut Properties**

Property	Description
Arguments	The arguments passed into the application. This is used when the shortcut points to an application (not a file or directory).
Description	This is a general description. Currently, it does not benefit a user. (You cannot view this description from the Explorer program.) This can be handy if you need to place a description of the shortcut's use for later reference.
File	This is the full path to the shortcut file. You should name this file with an <code>.LNK</code> extension (as in <code>C:\TEMP\MYFILE.LNK</code> ). UNC's are allowed.
Hotkey	This is a bitmask that represents a combination of keyboard keys (such as Control+Alt+E). This combination of keys will activate the shortcut.
IconLocation	A full path to a resource containing icons. Typically, this is a file containing icon resources such as a program, a DLL, or an icon file (such as <code>PERL ICO</code> ). UNC's are allowed.
IconNumber	This is the icon number found in the file pointed to by the <code>IconLocation</code> property.
Path	The full path to the object to which the shortcut will link. This is usually a program, a file, or a directory. UNC's are allowed.
ShortPath	This is a read-only property (you cannot change it). This property is the 8.3 version of the <code>Path</code> property. If you change the <code>Path</code> property, this will not change until you save and reload the shortcut.
ShowCmd	With this property, you can specify how the shortcut will display itself if launched from the Desktop or the Explorer program. <a href="#">Table 4.6</a> shows the options for this property.
WorkingDirectory	This is the working directory for the application specified in the <code>Path</code> property. If the <code>Path</code> is not an application but a file or a directory, this property is ignored.

**Table 4.6. Options for the `ShowCmd` property**

Option	Description
<code>SW_SHOWNORMAL</code>	Displays the result of the shortcut in a normal window.
<code>SW_SHOWMINNOACTIVE</code>	The result of the shortcut is minimized, but the currently active window remains active.
<code>SW_SHOWMAXIMIZED</code>	The result of the shortcut is maximized.

## Note

The Win32 API defines additional `SW_SHOW` constants to those listed in [Table 4.6](#). If you specify their values, however, the shortcut will replace the value with `SW_SHOWNORMAL`.

## Explaining the *Hotkey* Property

The `Hotkey` property requires a little bit of explaining—first in technical speak and then in English. The `Hotkey` value is derived from a 16-bit word (2 bytes). The most significant byte is the bitmask of the modifier keys, and the least significant byte is the virtual key code.

Okay, now for the explanation in English. If you look at the value in hexadecimal, you will notice 2 bytes. If the value of the `Hotkey` property is `1840` (decimal), for example, you can convert it to hexadecimal using the Perl `sprintf()` function, as in:

```
$Hex = sprintf( "0x%04x", 1840 );
```

If you print out `$Hex`, you will get `"0x0730"`. This shows that the decimal value `1840` is composed of 2 bytes: the first (most significant byte) is `0x07`, and the second (least significant byte) is `0x30`.

The first byte (`0x07`) represents a bitmask of the modifier keys in [Table 4.7](#).

**Table 4.7. Modifier Key Codes for the *Hotkey* Property**

Modifier Key	Hexadecimal Value
Shift Key	<code>0x01</code>
Control Key	<code>0x02</code>
Alt Key	<code>0x04</code>

You can determine which modifier keys are used by using the subroutine `GetModifier()` defined in [Example 4.6](#).

### **Example 4.6 Decoding the modifier keys**

```
01. my $Hotkey = shift @ARGV || 1840;
02. my( $KeyCode, $Modifier ) = unpack( "cc", pack( "S",
$Hotkey ) );
03. my %ModifierKeys = GetModifier( $Modifier );
04. print "The modifier keys for the hotkey $Hotkey are\n";
05. foreach my $Key ( keys( %ModifierKeys ) )
06. {
07.   print "\t$key\n" if ( ( $ModifierKeys{$Key} ) );
08. }
09.
10. sub GetModifier
11. {
12.   my( $Value ) = @_;
```

```

13. my( %Hash ) = (
14.     Alt => ($Value & 0x04)? 1:0,
15.     Control => ($Value & 0x02)? 1:0,
16.     Shift => ($Value & 0x01)? 1: 017.
17. );
18. return( %Hash );
19. }

```

The code in [Example 4.6](#) prints out the modifier keys for the bitmask of `0x07`. Line 2 uses a trick to programmatically discover the modifier and the keycode values. By packing the `Hotkey` property into a single, unsigned, short integer and then unpacking the values of the first and second bytes individually, we have broken the numeric value of `$HotKey` into the two separate values we need. An explanation of how this actually works is beyond the scope of this book and is left as an exercise for the reader.

Notice that lines 14–16 test the bitmask for the modifier keys. If the test determines that the modifier key is used, a value of 1 is assigned to the particular key; otherwise, 0 is assigned.

The second byte (`0x30`) is a *virtual key code*. This value corresponds to a key on the keyboard. This is the key you press (in addition to the modifiers) to activate the shortcut. [Table 4.8](#) provides the list of virtual key code values.

**Table 4.8. Key Codes for the Hotkey Property**

Keyboard Key	Hexadecimal Value
Backspace	0x08
Tab	0x09
Clear	0x0C
Enter	0x0D
Shift	0x10
Control	0x11
Alt	0x12
Pause	0x13
Caps Lock	0x14
Escape	0x1B
Spacebar	0x20
PageUp	0x21
PageDown	0x22
End	0x23
Home	0x24

**Table 4.8. Key Codes for the Hotkey Property**

Keyboard Key	Hexadecimal Value
Left Arrow	0x25
Up Arrow	0x26
Right Arrow	0x27
Down Arrow	0x28
Print Screen	0x2C
Insert	0x2D
Delete	0x2E
Help	0x2F
0	0x30
1	0x31
2	0x32
3	0x33
4	0x34
5	0x35
6	0x36
7	0x37
8	0x38
9	0x39
A	0x41
B	0x42
C	0x43
D	0x44
E	0x45
F	0x46
G	0x47
H	0x48
I	0x49
J	0x4A
K	0x4B
L	0x4C

**Table 4.8. Key Codes for the Hotkey Property**

Keyboard Key	Hexadecimal Value
M	0x4D
N	0x4E
O	0x4F
P	0x50
Q	0x51
R	0x52
S	0x53
T	0x54
U	0x55
V	0x56
W	0x57
X	0x58
Y	0x59
Z	0x5A
Left Windows key	0x5B
Right Windows key	0x5C
Applications key	0x5D
Numeric keypad 0	0x60
Numeric keypad 1	0x61
Numeric keypad 2	0x62
Numeric keypad 3	0x63
Numeric keypad 4	0x64
Numeric keypad 5	0x65
Numeric keypad 6	0x66
Numeric keypad 7	0x67
Numeric keypad 8	0x68
Numeric keypad 9	0x69
Numeric Keypad *	0x6A
Numeric keypad +	0x6B
Numeric keypad Enter	0x6C

**Table 4.8. Key Codes for the Hotkey Property**

Keyboard Key	Hexadecimal Value
Numeric keypad –	0x6D
Numeric keypad Num Lock	0x6E
Numeric keypad /	0x6F
F1	0x70
F2	0x71
F3	0x72
F4	0x73
F5	0x74
F6	0x75
F7	0x76
F8	0x77
F9	0x78
F10	0x79
F11	0x7A
F12	0x7B
Num Lock	0x90
Scroll Lock	0x91

## Creating a Hotkey

The section "[Explaining the Hotkey Property](#)" discussed how to interpret the `Hotkey` value. This section explains how to create a value that you can use to assign a hotkey.

Creating a `Hotkey` value is considerably easy. First, look through [Table 4.8](#) and pick out the value of the key you want to use. Suppose you want to use the PageUp key, which has a value of `0x21`:

```
$KeyCode = 0x21;
```

Next, pick the modifier keys you want to use. For the purposes of this example, you will use the Alt+Ctrl keys, whose values are `0x04` and `0x02`, respectively.

You need to logically OR the modifier keys together to get the modifier bitmask:

```
$Modifier = 0x02 | 0x04;
```

Now you need to create the `Hotkey` value. Use the reverse of the trick used in line 2 of [Example 4.7](#):

```
$Value = unpack( "S", pack( "CC", $KeyCode, $Modifier ) );
```

That is it! The variable `$Value` now contains the value of `1569`, which you will assign to the `Hotkey` property of your shortcut.

## Note

*On the Win32 platform, virtual key codes are not exactly the same as ASCII codes. It would be impossible to make keyboard codes match ASCII codes because a keyboard does not know the difference between the lowercase a and the uppercase A. The keyboard knows only that the key that represents the letter A has been pressed. If the operating system sees that the Shift key has been pressed at the same time as the A key, it knows that the capital letter A was intended.*

*What a virtual key code really represents is the numeric value that the keyboard sends to the computer, telling which key was pressed. [Table 4.8](#) lists these values.*

*The numeric keypad can generate different codes, however, depending on whether the Num Lock is active. The keypad's 7 key will generate either `0x24` if the Num Lock is not active, for example, or `0x67` if it is.*

## Creating a Shortcut

Creating a shortcut is quite easy and is accomplished by first creating a shortcut object:

```
$Shortcut = new Win32::Shortcut( [$Path] );
```

The only parameter (`$Path`) is optional. If a path to an already created shortcut file is passed in as the first parameter, the shortcut that the path points to is loaded. If there isn't a valid shortcut file, the `$Path` parameter is ignored.

If no parameter is passed in (or if the path passed in is not a valid shortcut path), an empty shortcut is created. You will have to set the properties of the shortcut (described later in this section); otherwise, it will not be functional.

If the function succeeds, a shortcut object is returned; otherwise, `undef` is returned. [Example 4.7](#) shows how to create a shortcut object.

## Note

*Creating a shortcut object does not commit the shortcut to disk. You must save the shortcut by calling the `Save()` method (see the section titled "[Saving the Shortcut](#)").*

## **Example 4.7 Creating a shortcut object**

```

01. use Win32::Shortcut;
02. my $File = shift @ARGV || die "Pass in a path to a .lnk file";
03. if( -e $File )
04. {
05.     my $Shortcut = new Win32::Shortcut( $File )
06.             || die "Could not create a shortcut object\n";
07.     print "The shortcut points to $Shortcut->{'Path'}\n";
08. }
09. else
10. {
11.     print "Could not find $File\n";
12. }<&I~files;shortcuts;creating>
```

## Modifying the Shortcut's Properties

After you have a shortcut object (refer to the "[Creating a Shortcut](#)" section), you can directly modify the shortcut properties. There are two methods to do this: modifying the members of the shortcut object (see [Table 4.5](#)) or using the `Set()` method:

```
$Shortcut->Set( $Path, $Arguments, $WorkingDirectory, $Description,
$Showcmd,
$Hotkey, $IconLocation, $IconNumber );
```

The `Set()` method requires all the parameters to be passed in; otherwise, whatever parameters are left off will default to 0 or an empty string (depending on the context of the property).

The `Set()` method will always return a 1, even if not enough parameters are passed in. Therefore, do not rely on error checking to determine whether you supplied enough parameters.

The parameters passed in are as follows (in order):

1. The `Path` property
2. The `Arguments` property
3. The `WorkingDirectory` property
4. The `ShowCmd` property
5. The `Hotkey` property
6. The `IconLocation` property
7. The `IconNumber` property

Refer to [Table 4.5](#) for descriptions of these properties.

The alternative to the `Set()` method would be to directly modify the properties. This can be done by modifying the property keys within the object (refer to [Table 4.5](#)), as shown in [Example 4.8](#). Pass in the path to a valid shortcut (a `.lnk` file) into the script, and both its description and working directory will be modified.

### ***Example 4.8 Modifying property keys within a shortcut object***

```

01. use Win32::Shortcut;
02. my $Shortcut = new Win32::Shortcut();
```

```

03. my $File = shift @ARGV;
04. if( $Shortcut->Load( $File ) )
05. {
06.     $Shortcut->{ 'Description' } = "My new description";
07.     $Shortcut->{ 'WorkingDirectory' } = "c:\\temp";
08.     $Shortcut->Save();
09.     $Shortcut->Close();
10. }

```

## Modifying and Retrieving Properties

There is an alternative to the procedures described in the "Modifying the Shortcut's Properties" section. The `Win32::Shortcut` extension defines a set of methods for both retrieving and setting the properties. The methods are as follows:

```

$Shortcut->Path( [$Value] );
$Shortcut->WorkingDirectory( [$Value] );
$Shortcut->Arguments( [$Value] );
$Shortcut->Description( [$Value] );
$Shortcut->ShowCmd( [$Value] );
$Shortcut->Hotkey( [$Value] );
$Shortcut->IconLocation( [$Value] );
$Shortcut->IconNumber( [$Value] );

```

Each of these methods can accept an optional parameter. If the parameter is passed in, the property is set to the value of the parameter.

All the methods return their respective property values. [Example 4.9](#) shows the use of the `Path()` method to both retrieve and set the `Path` property.

### ***Example 4.9 Using the property methods to alter a shortcut's path***

```

01. use Win32::Shortcut;
02. my $File = shift @ARGV;
03. my $Shortcut = new Win32::Shortcut( $File )
04.         || die "Unable to create a shortcut object.";
05. my $Location = $Shortcut->Path();
06. $Location =~ s/c:/d:/i;
07. $Shortcut->Path( $Location );
08. $Shortcut->Save();
09. $Shortcut->Close();

```

## Loading a Shortcut

After you have created a shortcut object, you can use it to load a shortcut file using the `Load()` method:

```
$Shortcut->Load( $File );
```

The first parameter is a path to a shortcut file. The path can be relative (to the current directory), a full path, or a UNC. The file specified must be a valid shortcut file.

If the `Load()` method is successful, the shortcut object will be filled with the properties from the shortcut file `$File` and a 1 will be returned; otherwise, `undef` is returned.

## Resolving Invalid Paths

If there comes a time when the target of a shortcut (the `Path` property) changes, the shortcut will need to be updated. This can be done either by changing the `Path` property and saving the shortcut or by using the `Resolve()` method:

```
$Shortcut->Resolve( [$Flag] );
```

There will be an attempt to resolve the `Path` property of the shortcut by verifying that it points to a valid path.

By default, the method works silently; however, if a 2 is passed in as `$Flag`, a GUI dialog box will be invoked, enabling the user to specify 2 new path if Win32 cannot resolve the path.

If the method is successful, the resolved path will be returned, and the `Path` property will have been updated; otherwise, an `undef` will be returned.

## Saving the Shortcut

If the shortcut object has been modified, you need to commit your changes to the hard drive by calling the `Save()` method:

```
$Shortcut->Save( [$File] );
```

The first parameter (`$File`) is optional, but if supplied, it will save the shortcut to `$File`. The value passed in can be a relative path (relative to the current directory), a full path, or a UNC.

If the file was saved successfully, the method returns a 1; otherwise, it returns `undef`.

## Warning

*No errors generate when creating or saving a shortcut over another file. You should make sure that whatever file you are saving the shortcut as is not some file you need.*

## Finishing with a Shortcut

After you have finished with your shortcut, you should use the `Close()` method:

```
$Shortcut->Close();
```

This method is not required but is good programming style. When this method is invoked, the shortcut object is closed and can no longer access its methods (including `Load()`). The `Close()` method always returns a 1.

## Note

*When the `Close()` method is called, the object is closed without saving the shortcut. Make sure you commit any changes by calling the `Save()` method before calling `Close()`.*

## Case Study: Modifying Shortcut Properties

I recently added a hard drive to my machine and moved over many data directories to the new disk. Because of this move, my shortcuts had to be altered. Because I have so many of them, however, I decided to use the script in [Example 4.10](#).

The code scans every shortcut in all directories of the current user's profile for a path that begins with `C:\Program Files` and replaces that path with `\Server\Share\Program Files`. It then prints how many shortcuts were scanned and how many were altered.

I have been able to use this script as the basis for many shortcut modifications. It is a rather handy piece of code.

### ***Example 4.10 Altering all shortcuts to point to a new path***

```
01. use vars qw( %Total $Shortcut );
02. use Win32::Shortcut;
03. my $Dir = $ENV{ "USERPROFILE" };
04. my %Path = (
05.     old => "C:\\Program Files",
06.     new => "\\\Server\\Share\\Program Files"
07. );
08. %Total = (
09.     changed => 0,
10.     checked => 0
11. );
12. # Let's escape our backslashes and other chars in
13. # the %Path hash since we will be using the values in a regex.
14. $Path{old} =~ s/([\\\\/.\\$^])/\\\\$1/g;
15.
16. #
17. # The meat of the script
18. #
19. $Shortcut = new Win32::Shortcut
20.         || die "Unable to create a shortcut object.";
21. ProcessDir( $Dir );
22. $Shortcut->Close();
23.
24. print "Total shortcuts checked: $Total{checked}\\n";
25. print "Total shortcuts changed: $Total{changed}\\n";
26.
```

```

27. #
28. #  ProcessDir() walks through a directory processing
shortcuts
29. #  and recursively calling ProcessDir() for other directories
30. #
31. sub ProcessDir
32. {
33.   my( $Dir ) = @_;
34.   my( @Dirs, @List, $File );
35.   print "\nDIR: $Dir\n";
36.   if( opendir( DIR, $Dir ) )
37.   {
38.     @List = readdir( DIR );
39.     closedir( DIR );
40.     foreach $File ( @List )
41.     {
42.       next if ( $File eq "." || $File eq ".." );
43.       my $Path = "$Dir\\$File";
44.       push( @Dirs, $File ) if ( -d $Path );
45.       ProcessShortcut( $Path ) if ( $File =~ /\.lnk$/i );
46.     }
47.     foreach $File ( @Dirs )
48.     {
49.       ProcessDir( "$Dir\\$File" );
50.     }
51.   }
52. }
53.
54. #
55. #  ProcessShortcut() processes the shortcut file. If it has a
path
56. #  pointing to the old path, change it to the new path.
57. #
58. sub ProcessShortcut
59. {
60.   my( $File ) = @_;
61.   my( $Path, $Name );
62.   $Total{checked}++;
63.   if( $Shortcut->Load( $File ) )
64.   {
65.     ( $Name ) = ( $File =~ /(^[\s\S]*)\.lnk$/i );
66.     print "\t$Name\n";
67.     $Path = $Shortcut->Path();
68.     if( $Path =~ s/^$Path{old}/$Path{new}/i )
69.     {
70.       $Shortcut->Path( $Path );
71.       $Shortcut->Save();
72.       $Total{changed]++;
73.       print " ****\t\t\t Changed ^^^\n";
74.     }
75.   }
76. }

```

## Tip

*There is an interesting thing about files on an NTFS partition: file streams. Files on an NTFS formatted drive are not the same as files on a FAT or HPFS partition. NTFS files consist of what are known as streams. Typically, a file is comprised of one stream. The stream is the collection of data that you think of as the file's data. For example:*

```
01. open( FILE, "> test.txt" ) || die "Error: $!";  
02. print FILE "This is a cool test\n";  
03. close( FILE );  
04. open( FILE, "< test.txt" ) || die "Error: $!";  
05. print <FILE>;  
06. close( FILE );|
```

*This will just write to a file and then print out its content:*

```
"This is a cool test"
```

*However, if you then run this code:*

```
01. open( FILE, "> test.txt:stream2" ) || die "Error: $!";  
02. print FILE "Holy cow, how can this be?\n";  
03. close( FILE );  
04. open( FILE, "< test.txt:stream2" ) || die "Error: $!";  
05. print <FILE>;  
06. close( FILE );
```

*You will see different output even though the same file is accessed:*

```
"Holy cow, how can this be?"
```

*If you think there is nothing nifty about this, that we have created two separate files, think again. Look at the directory listing and you will see only one file: test.txt.*

*Rename this file to testfile.txt and run this code:*

```
01. open( FILE, "< testfile.txt:stream2" ) || die "Error: $!";  
02. print <FILE>;  
03. close( FILE );
```

*You will see output of:*

```
"Holy cow, how can this be?"
```

*In other words, you have two separate data streams associated with a single file.*

## **Warning**

*The point of this exercise is that a file can have multiple data streams. If you move the file, the streams move as well because, as far as NTFS is concerned, all these data streams are just attributes (like the `readonly` and `hidden` flags) associated to a file. As a matter of fact, data in any NTFS file is considered to be an attribute (even if the data is hundreds of megabytes in size).*

*Macintosh users should feel at home with this concept because Macintosh files have always had two streams: the data and resource forks.*

## **Monitoring Directory Changes**

Changes to a directory can be a nightmare for administrators who need to know exactly what a directory contains. Suppose, for example, you have users who need to add and remove files from an FTP server's public directory tree. If each directory in the FTP site contains an index file enumerating which files are available, or if there are HTML pages on your Web server that provide links to each file in the directory, they need to be updated whenever someone changes the contents of the FTP directory tree.

There are a few ways to manage this:

- You could force all users to go through you or your staff when they add or remove files so that your staff can update these pages manually.
- You could have a process scheduled every night to go out and update the pages.
- You could use a CGI script or an ASP page to scan each of the directories to dynamically create the pages.

All three of these have some pretty massive drawbacks. The first suggestion could cause a time bottleneck and put quite a bit of strain on your staff. The second suggestion would give a large time resolution of updated files. If someone copies a file out to the directory tree for a client, it would not be listed until the next day. The third is the quickest and least stressful, but it would be processor intensive if you have a large number of users hitting the pages.

There is an alternative: the `Win32::ChangeNotify` extension. This extension does nothing but monitor a directory for changes. You can use this to detect a change that triggers a Perl script to run. This would be an ideal solution. First, of course, you must load the `Win32::ChangeNotify` extension:

```
use Win32::ChangeNotify;
```

There is another extension called `Win32::ChangeNotification`, which is the exact same as `Win32::ChangeNotify`. Evidently, someone decided the latter is easier to type, so it was renamed. You can use either one; just make sure you do not mix namespaces by using one extension but calling a function from the other. Additionally, you might not have both in your Win32 Perl installation, so check for the one you do have.

Before you can monitor a directory, you need to first create a change notification object. This is accomplished by using the `new()` method:

```
$NotifyObject = new Win32::ChangeNotify( $Path, $fSubTree,  
$Flags );
```

The first parameter (`$Path`) is the directory to be monitored. This can be either a full path or a UNC. Both forward and backslashes are supported as directory delimiters. This path must not end in a slash (or backslash); otherwise, it will fail. The only relative path that appears to work is the single dot current directory (`"."`).

The second parameter (`$fSubTree`) is a Boolean value that indicates whether only the specified directory is monitored or whether the directory and all its subdirectories are monitored. A `TRUE` value (`1`) indicates that all subdirectories are also monitored.

The third parameter (`$Flags`) represents flags that govern what is being monitored. This can be any combination of values from [Table 4.9](#) logically OR'ed together.

**Table 4.9. Change Notification Flags**

Flag Constant Name and its Short Constant Name	Description
<code>FILE_NOTIFY_CHANGE_ATTRIBUTES</code> <code>ATTRIBUTES</code>	Monitors all file and directory attributes, alerting when any change.
<code>FILE_NOTIFY_CHANGE_DIR_NAME</code> <code>DIR_NAME</code>	Monitors all directory names, alerting when a one changes.
<code>FILE_NOTIFY_CHANGE_FILE_NAME</code> <code>FILE_NAME</code>	Monitors all filenames, alerting when one changes.
<code>FILE_NOTIFY_CHANGE_LAST_WRITE</code> <code>LAST_WRITE</code>	Monitors the "last write" time stamps (the time that the file or directory was last written to or modified) on all files and directories, alerting when any of them change.
<code>FILE_NOTIFY_CHANGE_SECURITY</code> <code>SECURITY</code>	Monitors the security permissions on all files and directories, alerting when any change.
<code>FILE_NOTIFY_CHANGE_SIZE</code> <code>SIZE</code>	Monitors the size of a file or directory, alerting when any change. This will detect new files and directories that are added. Oddly enough, however, it will not alert if a file is deleted.

## Note

*For older versions of this extension, the `FindFirst()` function was used in place of the `new()` function. It took the same parameters, except that the very first one was a scalar variable that would be set with the value of the change notification object if the function was successful.*

*All the other parameters were shifted by one as in:*

```
Win32::ChangeNotify::FindFirst( $NotifyObject, $Path, $fSubDir, $Flags);
```

*Older versions of this extension worked identical to the newer version, except that all the methods have been renamed:*

<i>Old Name</i>	<i>New Name</i>
FindNext()	reset()
Wait()	wait()
Close()	close()

*Notice that all of the methods in the newer version of the extension consist of all lowercase characters.*

If the `new()` function is successful, it returns a valid `Win32::ChangeNotification` monitoring object. Otherwise, it fails and returns `undef`. [Example 4.11](#) shows how this function is used.

### **Example 4.11 Monitoring a directory (and subdirectories) for attribute or size changes**

```
01. use Win32::ChangeNotify;
02. my $Path = shift @ARGV || "$ENV{'SystemDrive'}\\\" ;
03. my $Flags = FILE_NOTIFY_CHANGE_ATTRIBUTES | 
FILE_NOTIFY_CHANGE_SIZE;
04. # create a change notification object
05. if( my $Monitor = new Win32::ChangeNotify($Path, 1, $Flags ) )
06. {
07.     print "Monitoring $Path...\n";
08.     while( $Monitor->reset() )
09.     {
10.         $Monitor->wait( INFINITE );
11.         print scalar localtime(), " There was a change to
$Path!!!\n";
12.     }
13.     $Monitor->close();
14. }
15. else
16. {
17.     print "Unable to create a change notification object.\n";
18.     print "Error:", $^E = Win32::GetLastError(), "\n";
19. }
```

After a monitoring object has been created, you need to indicate that you are monitoring it. This is done with the `reset()` method:

```
$NotifyObject->reset();
```

If `reset()` is successful, it returns `TRUE`; otherwise, it returns `undef`.

The `reset()` method takes no parameters and will start the process of monitoring. This method itself is not enough to provide a Perl script with the capability to know when a change has occurred. After monitoring is active, the script must wait for a change. This is done with a call to the `wait()` method:

```
$NotifyObject->wait( $Timeout );
```

The `$Timeout` parameter is the number of milliseconds to use as a timeout value. That is to say, this method will wait for up to the specified number of milliseconds. A value of `0` will immediately timeout, and a value of `INFINITE` will never timeout. In the latter case, the only way for the method to return is if the specified change occurs.

The `wait()` method will wait for the type of change to occur that was specified when the change notification object was created. If either a change or the timeout value is breached, the method returns.

`wait()` will return a `FALSE (0)` value *if a change has been detected*; otherwise (if the method times out), it returns a `TRUE` value (not `0` and not `undef`).

When a call to the `wait()` method returns, you can check the return value to determine whether it returned because of a timeout or because the sought-after change occurred. Either way, the script can continue monitoring by again calling `reset()`, which resets the monitoring, and then wait for another change by calling `wait()`.

After the monitoring has been completed, the change notification object needs to be destroyed and removed from memory. This is done by using the `close()` method:

```
$NotifyObject->close();
```

This will always return a `TRUE` value.

It is important to understand how this entire process works. After a change notification object has been created, any change to the monitored directory is placed in a queue. When you call `reset()`, it will clear the current state (if a change had already been detected by the `wait()` method, it is cleared) and advance the queue—literally finding the next change. A call to `wait()` will then check the queue. If it finds a change in the queue, it will return; otherwise, it will wait for a change to enter the queue and return (or return due to a timeout).

This entire process might seem a bit awkward, but it is important to understand because when your script returns from the `wait()` method, it will probably have to process some data. If during this time another change takes place, it will be placed in the queue. So the next call to `reset()` will reset the state and allow `wait()` to discover the next queued change. This prevents your script from missing a change because it was busy processing something.

The queue is only one element in length; so if 50 changes occur while your script is processing, it will only queue one of them. Really, that is all your script needs. There is no point in being told that 50 changes occurred because it will have to process the entire directory at once anyway.

It is interesting to know that if `wait()` returns because of a timeout, it can be called again. If it returns because it detected a change, however, the state must be cleared with a call to `reset()` before `wait()` will detect any new change. Any attempt to call it again without first calling `reset()` will return with a value indicating that a change has occurred even if no such change took place. The point here is that if `wait()` returns a `0` value (indicating a change has been detected), you must reset the current state by calling `reset()`.

If you need to clear the current state and any change that might be queued up, you can call `reset()` twice. By doing so, you reset the current state and advance the queue. Doing so twice will, in effect, clear the current state as well as the queue.

[Example 4.12](#) is a good example of how change notification could be used. It accepts a directory path as a command-line parameter.

## Note

*Some versions of Win32::ChangeNotify (and Win32::ChangeNotification) did not export the INFINITE constant. Because this is one of the most commonly used constants from this extension, it would be wise to manually add it to the list of exported constants from the CHANGENOTIFY.PM file.*

*You could always use the constant's value instead, which is 0xFFFFFFFF.*

### **Example 4.12 Report any size changes to a particular directory**

```
01. use vars qw( $fSubDirs $STAT_FILE_SIZE );
02. use Win32::ChangeNotify;
03. # Define the $fSubDir global variable...
04. $fSubDirs = 1;
05. # This is the index of a value returned by the stat() function.
06. $STAT_FILE_SIZE = 7;
07. my $Path = shift @ARGV || "$ENV{'SystemDrive'}\\";
08. my $Flags = FILE_NOTIFY_CHANGE_FILE_NAME |
FILE_NOTIFY_CHANGE_SIZE;
09. print "Setting up to monitor '$Path'...\n";
10. my $Monitor = new Win32::ChangeNotify( $Path, $fSubDirs,
$Flags )
11. || die "Could not create a notification object. ", Error(),
"\n";
12. my $FileList = ScanDir( $Path );
13. while( $Monitor->reset() )
14. {
15.   my $Changes;
16.   print "\nMonitoring '$Path' ";
17.   print (( $fSubDirs )? "(and sub directories)":"" );
18.   print "... \n";
19.   $Monitor->wait( INFINITE );
20.   print "\t-Changes detected at " . localtime() . "\n";
21.   ( $FileList, $Changes ) = DetectChanges( $FileList, $Path );
22.   if( scalar( keys( %$Changes ) ) )
23.   {
```

```

24.     my $iCount = 0;
25.     foreach my $File ( sort( keys( %$Changes ) ) )
26.     {
27.         $iCount++;
28.         print "\t$iCount) $File $Changes->{$File}.\n";
29.     }
30. }
31. else
32. {
33.     print "\t- No noticeable changes occurred.\n";
34. }
35. }
36. $Monitor->close();
37.
38. sub DetectChanges
39. {
40.     my( $OldList, $Path ) = @_;
41.     my( $NewList ) = ScanDir( $Path );
42.     my( $Changes, $File );
43.     foreach my $File ( sort( keys( %$OldList ) ) )
44.     {
45.         my $Delta = $NewList->{$File} - $OldList->{$File};
46.         if( $Delta )
47.         {
48.             if( ! -e $File )
49.             {
50.                 $Changes->{$File} = "has been removed";
51.             }
52.             elsif( $Delta < 0 )
53.             {
54.                 $Delta = abs( $Delta );
55.                 $Changes->{$File} = "has been reduced by $Delta bytes";
56.             }
57.             else
58.             {
59.                 $Changes->{$File} = "has grown by $Delta bytes";
60.             }
61.         }
62.     }
63.     foreach $File ( sort( keys( %$NewList ) ) )
64.     {
65.         if( ! defined $OldList->{$File} )
66.         {
67.             $Changes->{$File} = "has been added";
68.         }
69.     }
70.     return( $NewList, $Changes );
71. }
72.
73. # Generate a hash of files and their sizes
74. sub ScanDir
75. {

```

```

76. my( $Path, $List ) = @_;
77. my( @Entries, $Temp );
78. if( opendir( DIR, $Path ) )
79. {
80.     @Entries = readdir( DIR );
81.     closedir( DIR );
82. }
83. foreach my $Temp ( sort( @Entries ) )
84. {
85.     next if( $Temp eq "." || $Temp eq ".." );
86.     my $Dir = "$Path\\$Temp";
87.     if( -d $Dir )
88.     {
89.         ScanDir( $Dir, $List ) if( $fSubDirs );
90.     }
91.     else
92.     {
93.         $List->{ $Dir } = FileSize( $Dir );
94.     }
95. }
96. return( $List );
97. }
98.
99. # Return the size of a file
100. sub FileSize
101. {
102.     my( $Path ) = @_;
103.     return( ( stat( $Path ) )[ $STAT_FILE_SIZE ] );
104. }
105.
106. sub Error
107. {
108.     return( $^E = Win32::GetLastError() );
109. }

```

## Summary

Files can contain a wealth of information that we take for granted. Not only does the data contained within the file have significant value, so does the file itself. The file's attributes tell much about the state of a file. In addition, file information, such as the original name of the file, can be of great interest to anyone (especially administrators).

The Win32 Perl extensions provide functions that empower administrators to manage files and directories by giving access to the attributes of both. In addition, with file systems that provide extended attributes, file and directory attributes are available to a Perl script as well.

File attributes are taken for granted by most users. Users generally don't care whether a read-only bit is set or the archive bit has been turned off— just as long as everything works. When an administrator needs to set a configuration file as read-only or check whether a file is compressed, however, he appreciates the `Win32::File` extension.

It is not often that a user needs to know the version number of a particular `.DLL` file. Administrators do need to know this, however, to keep a machine in sync with others. Using the `Win32::AdminMisc` extension opens the door to accessing a plethora of file information.

Shortcuts are being used more and more by users, and they too have to be managed. These files can be managed by using the `Win32::Shortcut` Win32 Perl extension. Managing these shortcuts is one of the problems that impacts administrators, but most don't know about it until it becomes a need.

## Chapter 5. Automation

One of the most useful aspects of a Win32 machine is the capability to embed and automate data and programs. Microsoft has accomplished this by implementing the complex technology known as object linking and embedding (OLE). Just as it is strikingly similar to the UNIX-based CORBA, it is also just as powerful. Therefore, many users have learned to take advantage of OLE in such a way that it is no longer considered to be a separate technology; instead, it is assumed to be an imperative part of the operating system.

This chapter describes how a Perl script can make use of this technology by using the `Win32::OLE` extension.

### Understanding OLE

In early versions of 16-bit Windows (such as version 3.1), Microsoft introduced a new concept called object linking and embedding (OLE). The idea behind this new technology was that you could embed an object within another object. If you were sending an email message using Microsoft Mail, for example, you could paste a copy of a Word document into it. This was very cool because the receiver could just double-click on the attached document (which was displayed as an icon). That would tell your word processor to load the document automatically. If the word processor were not already running, it would first be launched and would then load the document. Linking objects with other objects was a great idea, but OLE itself was quite limited. Originally, OLE used a technology called dynamic data exchange (DDE) as the protocol to communicate between programs.

DDE was the mechanism used to exchange information between different applications. For example, an Excel spreadsheet could use DDE to send data to a word processor. Another example is how an application would tell the old Windows Program Manager (the predecessor to Explorer) to create a new Program Group icon. The program would have to open a DDE channel with the Program Manager and submit commands and data. The Program Manager would know how what to do with these commands and data, such as create, modify, or delete an icon. This is how setup programs would create a group and populate it with icons that launch the program.

The DDE technology eventually "matured" into a network-aware version called NetDDE. This simply enabled an application to send DDE commands to remote machines.

From a technical standpoint, DDE and NetDDE were a mess. You might have to submit a command several times before the target application actually accepted it. It was also slow, unreliable, and inefficient. As a first attempt it was okay, but it quickly fell out of favor for another technology. However, to retain backward compatibility, DDE exists even today in Windows 2000/ME.

Several years after OLE was introduced, OLE 2.0 was released. This replaced DDE with a new protocol called the component object model (discussed later in the section titled "[COM](#)") as the

communication mechanism. OLE 2.0 supported enhancements that made it easy and a pleasure to use. OLE 2.0 provided the capability to edit embedded data within a document. OLE enables a user to paste an Excel spreadsheet into a Word document, for example, but editing the spreadsheet with OLE 1.0 would cause Excel to run and load the spreadsheet. The user would then have to switch to the running Excel application, edit the spreadsheet, save the changes, and then quit Excel. OLE 2.0, on the other hand, fostered a sense of integration. To edit the embedded spreadsheet, the user just placed the cursor into the spreadsheet that appeared in the Word document. At that point, OLE 2.0 merged Word's menus and toolbars with Excel's menus and toolbars. This way, a user could edit the Excel spreadsheet from within the Word program. The capability to activate one program's functionality from within another is called *in place activation*.

OLE 2.0 also brought about a new concept called *automation*. This is where one program can tell another application what to do. Imagine a user who needs to send a letter to everyone in a database of names and addresses. He could perform a query on his database and then use automation to load his word processor, create a form letter inserting the names and addresses retrieved from the database, and then tell the word processor to print out the letters.

Before going into the explanation of how Perl can use OLE and COM, it might be of use to explain exactly what OLE is and how it works. There are many terms to define and acronyms to explain so that OLE makes sense, especially because the industry has been using the terms OLE, COM, and ActiveX synonymously. So first, familiarize yourself with some basic terms.

## **OLE**

Object linking and embedding (OLE) is the overall technology that enables one process to control or talk to another process. An OLE transaction consists of a client process, a server process, and a communication protocol that the processes use to talk with each other.

## **COM**

The component object model (COM) is a protocol used to communicate between processes involved in an OLE transaction. This protocol replaced DDE as the communication method of choice with the introduction of OLE 2.0. All Win32 processes that make use of OLE use COM to transfer data and send instructions to each other. Win32 still supports DDE for backward compatibility reasons, but OLE no longer uses it.

When a program wants to talk to another program, it must follow the rules of conduct prescribed in the COM protocol. Consider COM to be the "rules of the road" for programs to chat with each other. By following these rules, each program knows what to expect from another program while using COM to talk with each other. For all practical purposes, a Perl programmer needs only to know that there exists a thing called COM and it is related to OLE. The actual OLE programs (such as Excel and Word) take care of the rest.

## **ActiveX**

An ActiveX component is just a COM-based server (described in the next section, "How OLE Works") that follows certain rules that all ActiveX components must follow. These rules enable the component to be embedded in other applications such as HTML pages and Web browsers. An ActiveX component can be accessed from Perl as long as it supports what is known as the IDispatch interface (described later in the section "Interfaces") and does not need to be instantiated within something called a control container. Some ActiveX components are designed to be "controls" like

a dial that indicates how loud a stereo is. These controls are designed to be embedded inside an application, not run as a standalone component. Therefore, they require some other *container* object that would embed the control. As of Perl 5.6, it does not act as a control container, so any ActiveX component you need to interact with must not require a control container.

## How OLE Works

When a program wants to talk to another program, it can use OLE (or more accurately, COM) to strike up a conversation. The program (or process) that starts the conversation is called a *controller* because it is beginning the process of controlling the other application. Any program or process that a controller talks to is known as a COM *server* because it will be serving the controller. Programs that act as servers come in two flavors: in-process and out-of-process servers.

Simply stated, an *in-process* server (also known as an *inproc* server) is just a DLL file that the controller process loads into memory and uses. It is called an in-process server because a DLL file is loaded into the same process (or memory) space as the application loading it. An example could be a fax printer driver. If your application wanted to send a fax, all it would have to do is load the fax printer driver's DLL and make calls into the DLL directly.

An *out-of-process* server (sometimes called a local server application) is a separate executable program run when the controller begins a conversation with it. After the local server application is running, the controller process talks to it using the COM protocol. An example of this is a program such as Microsoft Excel. When Excel is loaded as a local server application, it literally is started up as if a user had double-clicked on its icon. Some applications will run but hide their main window so that users are unaware they have started (Microsoft's Excel and Word behave this way).

If DCOM (distributed COM) is installed, the out-of-process server could be on another machine, and communication could be done over a network. This is cool because the server application will physically run on another computer. If this were the case, the out-of-process server would be called a *remote server application*.

For the most part, knowing whether a COM server is in-process or out-of-process is not very important to the average Perl coder. Either way, Perl interacts with the COM servers the same way.

Now consider a real-life scenario. You are in your small business office, and you need to fax out form letters to your customers. Each letter needs to have personalized information about a customer's account and a graph illustrating his yearly interactions with your business. So you write a form letter using Microsoft Word. You then need it to connect to an Access database and query each customer's data. The data is held on the accounting computer, so Word starts a COM conversation to Access on that machine. Access starts running on the remote computer (hiding its window so the accountant can't see it, of course) and processes Word's request for information. Here Access is acting as a remote out-of-process COM server.

Access sends Word the requested data. Word then starts a conversation with Excel to graph the data. The Excel program is located on the same machine as Word, so it is considered to be a local out-of-process COM server. Excel begins to run (again, hiding its windows) and generates a graph from the data that Word received from Access. When the graph has been generated, it is placed in the form letter along with the other client data.

Finally, Word prints the letter to the fax device. This causes a fax printer driver (a COM-based DLL file) to load into the same process space as Word itself (as all printer drivers do). Word interacts

with the DLL by calling COM interfaces on it. The fax printer driver processes the request, and the letter is faxed to the client. In this case, the fax printer driver is acting as a in-process COM server. So there you have it: the Microsoft Office Suite interacting using various COM techniques.

## Using OLE in Perl

Suppose you want to place data from your application into an Excel spreadsheet. You would want to first create an Excel COM object. Of course, this process is called *automation*, and COM is just the communication protocol. The fine folks in Microsoft's marketing department are pushing COM to be the acronym *du jour*, so you need to create an Excel COM object. You then interact with the object, and after you are finished with it, you just destroy it. Now, isn't that simple? The trick here is knowing what to do with the COM object. Generally speaking, you can examine it, alter it, and tell it to do things.

## COM Objects

To explain exactly what a COM object is, we must first go off on a tangent and talk about something that might be a little bit easier to understand: those wonderful ATMs (automated teller machines). For you to use an ATM, you need to first go to your bank and get an ATM card. When armed with this piece of plastic, you can not only query the bank's database and get the balance of your accounts, you can also tell the bank to perform transactions such as deposit, withdraw, or transfer money from account to account.

COM objects are similar to the ATM card. Each COM object has a particular set of values, known as *properties* (such as your balance), and functions, called *methods*, that perform a task (such as deposit and withdraw). After you have a COM object, you can check its properties to query the state of the object, change properties altering the object's state, or tell the object to perform some function. Examples of properties could be the font of a particular spreadsheet cell or the filename of a document. Examples of methods could be `Open()`, `Print()`, or `Quit()`.

## Identifying COM Objects

Okay, suppose you decide you want to write code that interacts with a COM object. You now need to determine what *type* of COM object you are going to interact with. If you have two versions of Word (Office 97 and Office 2000) installed on your machine, for example, you need to make sure that you are interacting with the correct version. Likewise, you will need to know how to interact with Excel rather than Word.

Every COM object is based on a class. A class is like a blueprint that describes how an object will behave. This class is just like an object-oriented Perl, Java, or C++ class. An OLE-aware program typically consists of several classes. Consider a COM server that exposes several different classes, such as Microsoft Excel. It consists of many classes such as application, workbook, worksheet, and graph classes.

Excel defines each of these classes and exposes them so that a controller application (like your Perl script) can request any of them to be instantiated into COM objects that you can interact with.

Some classes rely on others. If you just want to create a new Excel worksheet, for example, you could request a worksheet object. To have a worksheet, however, Excel must first be started and a new workbook created—often (but not always), this is done automatically by the server application.

The resulting COM object would only represent the worksheet, even though both Excel and a workbook have also been created.

## **GUIDs**

Each OLE class has a *global unique identifier* (GUID, pronounced "goo-id" and rhymes with "grid") that no other class shares. These are those funky-looking strings found throughout the Registry. All GUIDs follow a specific format that looks like the one that follows (which represents the Excel 9.0 application that comes with Office 2000):

{00024500-0000-0000-C000-00000000046}

It is important to understand that GUIDs are hard coded and do not change. A GUID for a class on Joel's computer will be identical to the one for the same class on Jane's computer. All computers that have MS Access 2.0 installed will have the same GUID. If you have MS Access 9.0 installed, you will have a different GUID. If you have both versions installed, you will have both GUIDs in your Registry. The point here is that you can depend on a particular GUID to always represent a particular program.

### **Note**

*When a programmer writes an OLE-aware application, she needs to create a GUID for it. Typically she will run Microsoft's [GUIDGEN.EXE](#) application, which generates shiny new GUIDs.*

*Because of the mathematical probability involved with the creation of GUIDs, it is unlikely that any two GUIDs will ever be the same. For all practical purposes, it is so unlikely that it is considered not possible. This, of course, assumes that you use the Microsoft-approved, GUID-generating algorithm that is a part of the OLE API (which [GUIDGEN.EXE](#) indeed does use).*

A GUID is often referred to as a CLSID (class identifier), an AppID (application ID), an IID (interface ID), or a UUID (universally unique ID).

## **Class Names**

Class names are implemented because a GUID is quite long and difficult for humans to understand and remember. A class name is just a common name used to reference the GUID.

When you use OLE, you can specify the GUID {73FDDC80-AEA9-101A-98A7-00AA00374959}, or you can instead use the class name `Wordpad.Document.1`. They both refer to the same OLE class, but one obviously makes it easier for humans to understand the program it represents.

Usually a class name will be some reasonable, understandable text string. For example, Microsoft Excel has an `Excel.Application` class name. You can use this to communicate with Excel. However, if you have multiple versions of Excel installed (let's say Office 2000 and Office XP) on your machine, then using `Excel.Application` does not indicate which version of the program you are referring to. Therefore, the class name `Excel.Application` simply points to another class, such as `Excel.Application.9`.

You can use either of the classes. Since one points to the other, they both cause Excel version 9.0 to run. However, if you use the class name "Excel.Application.10", then the Excel version 10 program would run. This is irrespective of "Excel.Application" pointing to version 9.

## **Interfaces**

Another very important aspect of COM is what is known as an *interface*. An interface is a set of methods and properties. They are typically somehow related to each other. Usually an interface is simply the methods and properties that a class exposes.

When your Perl script has a COM object, it really has a pointer to an interface. It can then query the interface for properties, set properties on the interface, and call methods on the interface. The GUIDs previously discussed are really identification numbers (UUIDs—universally unique identifiers) that indicate a particular interface.

All COM objects must expose at least one interface, known as *IUnknown*. The three methods that this interface contains are the very basis of COM. For any COM object that is automation compliant (and hence is capable of being controlled from a scripting language such as Perl and Visual Basic), it must expose an additional interface called *IDispatch*.

The *IDispatch* interface contains a few methods that automation requires. These enable an automation language to learn what properties and methods exist on the COM object. This capability of discovery is essentially the core of what makes automation possible.

The particular methods that *IUnknown* and *IDispatch* require are not really important to know for coding Perl. The `Win32::OLE` extension manages all the interaction between Perl and these methods for you.

## **Perl and OLE**

Win32 Perl originally had the capability to handle OLE transactions by means of the original `OLE.PM` module. This module has become obsolete and has been replaced with the `Win32::OLE` extension by Gurusamy Sarathy and Jan Dubois. Although the fundamentals between the two are the same, the implementations are somewhat different, and in some cases, the original `OLE.PM` is just not designed to handle what `Win32::OLE` can.

For the most part, this chapter describes `Win32::OLE`—even though the extension is not compatible with ActiveState versions before 5.005.

### **Tip**

*If you have scripts that make use of the `OLE.PM` module, you should consider migrating to the `Win32::OLE` extension. It provides much more flexibility and is less prone to runtime errors.*

## **Creating COM Objects**

Now that OLE and COM have been briefly described, forget everything regarding OLE. It is not important. OLE is simply the way in which you can embed an object into another object. OLE defines how that occurs. When you really dig deep into the specifics, you will find that OLE *really*

represents how you can embed a COM object inside another COM object. Add emphasis on COM object.

If you are slightly confused, then you are ready for this next piece of information. A Perl script will be playing with COM objects, not OLE. It is unfortunate that the extension is called `Win32::OLE` because it does not really provide OLE capabilities to Perl. It provides access to COM objects. I suppose if it were correctly called `Win32::COM`, it would be confused with some extension that exposes serial ports on a Windows machine.

So there it is: The `Win32::OLE` extension provides access to COM objects.

When a Perl script wants to interact with a COM object, the script is considered to be a controlling process. The Perl script needs to first access a COM object. It is important to understand that the Perl script does not create the COM object; instead, the COM server (that is, the COM-based application with which the script is to interact —this is sometimes called a *COM server*) creates the object on the Perl script's behalf. After that object has been created, the `Win32::OLE` extension returns a Perl object that represents the COM object. Really what is returned to the Perl script is a special object that points to an instance of a COM object's interface. Using this special Perl object, the script can interact with the COM object.

Your Perl script needs to obtain a COM object from an OLE server. There are several different ways to do this because there are several different ways to obtain the object. You can request that the COM server create the object, you can try to find an existing object, or you can try to load a persistent object (typically, this is stored as a file somewhere).

### Note

*Many users of `Win32::OLE` (and the original `OLE.PM`) module think that the module creates the COM object. To understand COM, you must understand that the Perl script never creates a COM object; it requests that the object be created. This distinction is very important.*

*If the Perl script could create a COM object, it would be considered a COM server application, which it is not. Because the Perl script is just a controller application, it can only request that server applications create a COM object.*

## Creating a New COM Object

For a Perl script to interact with a COM object, it must first create a `Win32::OLE` object, which in turn requests that the OLE server application create a COM object. This is what the `new()` method is used for. By using this method, you are requesting that the server application create a new COM object for your use. There are two ways to use `new()`. The first listed is the more conventional way, but they both work:

```
$Object = new Win32::OLE( $Class [ , $DestroyCommand ] );
$Object = Win32::OLE->new( $Class [ , $DestroyCommand ] );
```

These two techniques are equivalent to each other. Which one your script uses is totally up to your whim. I prefer the former because it more accurately mimics C++'s object creation semantics.

The first parameter (`$Class`) is a text string that represents a registered COM class. This is either the class' GUID, or its name.

If you are specifying a DCOM (distributed COM) connection to a remote machine, the first parameter can be constructed as an array reference

```
$Class = [ $$Machine, $ClassID ];
```

where `$Machine` is a valid computer name such as `\MyMachine` or `mymachine.mydomain.com`.

By specifying an array reference, you are telling the `Win32::OLE` extension that you want to use DCOM (distributed COM). If your local machine and the remote address you specify both have DCOM installed, an attempt to access a remote COM object will take place. Your script will need to run under a user account that has DCOM access permissions for the remote machine.

The second parameter (`$DestroyCommand`) is optional and is either a string or a code reference. It represents either a method of the COM object or a subroutine that your script defines, and it will be called when the Perl `Win32::OLE` object is destroyed. Unfortunately, some COM objects do not automatically unload from memory when they are destroyed. This means that when a Perl script terminates, some COM objects might remain floating around in memory. (This is a big problem with out-of-process server objects—you might end up with many copies of a server application remaining in memory.) To avoid this, a `Win32::OLE` object will call whatever method is specified as the second parameter when it is destroyed. Typically, you specify the method to quit the OLE application. For an Excel object, you would want to specify `Quit` so that the `Quit()` method will be called when the `Win32::OLE` object is destroyed.

If the `$DestroyCommand` parameter is a reference to a code segment, that code is called when the `Win32::OLE` object is destroyed. The `Win32::OLE` object (which is being destroyed) is the only parameter passed into the specified code segment. The routine can use this to interact with the COM object.

If the `Win32::OLE` object is successfully created, a `Win32::OLE` Perl object is returned; otherwise, `undef` is returned. If a DCOM object was requested, then any object that is returned represents the COM object on the specified remote computer. You treat it as if it were a normal, local COM object.

When specifying the class ID (the first parameter), you can use either a GUID or a class name. If you specify a class name, Win32 will just look up the class's GUID from the class database (located in the `HKEY_LOCAL_CLASSES` hive in the Registry). The main reason for using a class name is that it is easier for humans (in particular a programmer) to recognize.

## Warning

*Keep in mind that the use of the `new()` method will request that a new COM object be created. This can have substantial memory and processor overhead. If you were to request an Excel COM object by using the `new()` method, for example, Win32 would have to start an instance of the Excel application.*

*If a script requested several Excel COM objects using the `new()` method, the result would be several instances of the Excel application running. This would quite quickly cause the script to run*

*out of memory and incur a radical performance hit. Therefore, using the `new()` method would not be a wise choice for a CGI script.*

*Typically, COM objects created using the `new()` method are not visible on the screen. Regardless, they are indeed running and taking up memory. Line 18 in [Example 5.1](#) sets the `Visible` property to 1 so you can see that, indeed, an instance of Excel is running. If you were to run this code five times simultaneously, you would see that five instances of Excel would be started.*

If it is possible that several instances of a COM object will be necessary, it would be wiser to request the first one using the `new()` method and the remaining ones using the `GetActiveObject()` method instead, as in [Example 5.1](#).

### **Example 5.1 Creating and interacting with a COM object**

```
01. use Win32::OLE;
02. # Set some variables
03. my $Class = "Excel.Application";
04. my $File = "c:\\temp\\MyTest.xls";
05.
06. # If the file exists already then delete it
07. unlink( $File ) if (-e $File);
08.
09. # If you can use an already running copy of Excel
10. my $Excel = Win32::OLE->GetActiveObject( $Class );
11. if( ! $Excel )
12. {
13.     $Excel = new Win32::OLE( $Class, \&QuitApp )
14.     || die "Could not create a COM '$Class' object";
15. }
16.
17. # Show the Excel object so we can see it
18. $Excel->{Visible} = 1;
19. $Excel->{SheetsInNewWorkbook} = 1;
20.
21. # Add (or create) a new workbook
22. my $Workbook = $Excel->Workbooks->Add();
23.
24. # Select the first worksheet in the workbook
25. my $Worksheet = $Workbook->Worksheets(1);
26. $Worksheet->{Name} = "My Test Worksheet";
27. sleep(1);
28.
29. # Change the contents of a "range" of cells ((actualy only one
cell))
30. my $Range = $Worksheet->Range("A1");
31. $Range->{Value} = "Hey, this is a test!";
32. sleep(1);
33.
34. # Save the workbook
35. $Workbook->SaveAs( $File );
```

```

36. sleep(1);
37.
38. sub QuitApp
39. {
40.     my( $ComObject ) = @_;
41.     print "Quitting " . $ComObject->{Name} . "\n";
42.     $ComObject->Quit();
43. }

```

In [Example 5.1](#), a class name is used (`Excel.Application`) to create the COM object. The server application's GUID (`{00024500-0000-0000-C000-00000000046}`) could be used instead. Also notice that a destroy method is specified (as a reference to a code segment). The reason is that, if the COM object is created on our behalf (by the `new()` method in line 13), the code is responsible for making sure the COM object is terminated when it is finished. Most COM objects are thoughtful enough to know when they are no longer needed and terminate themselves. Microsoft Office 97 is, evidently, not one of them, so we must make sure that Excel's `Quit()` method is called. Even if you are referencing an application that does the right thing, specifying a termination function would not hurt. When the `$Excel` object is destroyed (even when the script terminates early from, for example, pressing Ctrl+Break), the `Win32::OLE` object will call the Perl subroutine specified as the second parameter of the `new()` method (line 13). In this example, when the `Win32::OLE` object is destroyed, the object will call the `QuitExcel()` subroutine and pass in a reference to the Perl object being destroyed. The routine will explicitly call the COM object's `Quit()` method.

If you changed line 13 of [Example 5.1](#) to

```

$Excel = new Win32::OLE( [ "www.foo.com", $Class ], \&QuitExcel )
|| die "Could
not create an OLE \"$Class\" object";

```

`Win32::OLE` would use DCOM to request the COM object from the computer [www.foo.com](http://www.foo.com). This means that, if successful, the Excel server application would be running on the remote machine ([www.foo.com](http://www.foo.com)) rather than your local computer. This can be very handy if you have a powerful application server that can handle the load of several instances of an application running. It can also be a devastating prank if you target the computer of a coworker who is not aware of what you are doing.

## Warning

*Some computers have problems with instantiating COM objects after a machine has come out of hibernation or standby mode. Calls to instantiate COM objects (for example, calling `new()`, `GetActiveObject()`, and `GetObject()`) have been known to fail. They might return a `Win32::OLE` object, but the object does not accurately represent the requested COM object.*

*Under these circumstances, rebooting the computer has been known to alleviate the problem.*

Using DCOM for delegating power servers to handle many instances of Excel or MS Access, for example, can be quite inviting. You might have a Web server that receives several simultaneous hits

on CGI scripts or ASP pages that request COM objects, for example. By using DCOM, you can keep your Web server from bogging down under the load of COM object requests. Keep in mind, however, that this would come at a cost of bogging down other machines (which you would have running the COM server applications) and increasing network traffic.

Creating a COM object is quite often necessary, but it can be more practical (in both memory and processor usage) to use existing objects. This is why the `GetActiveObject()` method exists.

## Note

*Numerous articles and books have been written that suggest using the `CreateObject()` method. Although this will work the same as the `new()` method, it is only provided for backward compatibility with previous versions such as the older `OLE.PM` module.*

*It is highly recommended that you use the `new()` method rather than `CreateObject()`. There is no guarantee that `CreateObject()` will be supported in future releases.*

## Accessing an Existing COM Object

An alternative to `new()` is the `GetActiveObject()` method. Whereas `new()` will request that a COM object be created, `GetActiveObject()` will return a reference to an object that already exists:

```
$Object = Win32::OLE->GetActiveObject( $ClassID [,  
$DestroyCommand ] );
```

The first parameter (`$ClassID`) is a string representing the class ID. This ID can either be a GUID or a class name. Unlike the `new()` function, you cannot supply a two-element array reference for this parameter because DCOM is not supported by the `GetActiveObject()` method. This function can only be used to obtain a local COM object.

The optional second parameter is the same as the optional second parameter for the `new()` function. It is either the name of a function or a function reference. This specified function will be called when the resulting COM object has been destroyed. This parameter is not available in older versions of the `Win32::OLE` extension.

The function returns a `Win32::OLE` object that represents the specified type of COM object only if such an object already exists in memory. Otherwise, the function fails and returns `undef`.

Note that, unlike the `new()` method, `GetActiveObject()` will neither start the server application nor create an instance of the specified COM object. It is a good idea to first use this method in a script. If it fails, use the `new()` function. This way, if Excel is already running your script, for example, it will not force another instance of Excel to run. This will save on memory and processor usage.

Line 10 in [Example 5.1](#) demonstrates the use of the `GetActiveObject()` function. If the function fails to find a COM object, the code will call the `new()` function and request that a COM object be created.

## Accessing a Persistent COM Object

Yet another way to obtain a COM object is to request one based on a *persistent object*. A persistent object is considered to be a COM object that has been saved, typically as a file on a hard drive. If you are editing a Microsoft Word document and then save it as `C:\FILES\MYFILE.DOC`, for example, this file is considered to be a persistent object because, when you load it back into Word, it is in the same state as it was when you saved it. You could consider the document to be kind of a snapshot of the word processor the last time you saved the document. If you consider the document to be an object (which COM does), you could say that the file represents a saved state of the object. This is known as a persistent object; however, most people think of them as simple Word, Excel, or some other application document files.

The `Win32::OLE` extension enables you to access persistent COM objects, such as a file, by using the `GetObject()` function:

```
$Object = Win32::OLE->GetObject( $Path [, $DestroyCommand ] );
```

The first parameter (`$Path`) is the path to a persistent object (like a `.DOC` or `.XLS` file). This path can be a relative, full, or UNC path. It can also be an URL.

This path is called a *moniker*. It can contain an optional named subcomponent by appending an exclamation point (!) and the name of the object. Suppose you need to obtain an object that represents a particular worksheet called "Totals" in a workbook. You can specify the path to the workbook file concatenated with "`! Totals`". The path passed into the `GetObject()` method would look like the following:

```
$Worksheet = Win32::OLE->GetObject( "c:\\temp\\mytest.xls!Totals" );
```

Monikers that contain subcomponents like this are called *composite monikers*. Such monikers can consist of multiple subcomponents. For example, if you wanted to obtain a COM object for just a few cells in a particular worksheet, you might use the moniker:

```
"c:\\temp\\mytest.xls!Totals!A1:C3"
```

The resulting object would reflect the workbook's specified subcomponent (the "Totals" worksheet).

This is how it is *supposed* to work; however, just as in life, things don't always work as advertised. It is up to the application (the COM server) to recognize subcomponents in a moniker. The preceding example will fail when used with Microsoft's Excel application. You might have to experiment with applications (or read the documentation) to decide whether they support such composite monikers.

The optional second parameter is the same as the optional second parameter for the `new()` function. It is either the name of a function or a function reference. This specified function will be called when the resulting COM object has been destroyed. This parameter is not available in older versions of the `Win32::OLE` extension.

If the `GetObject()` method is successful, it returns a `Win32::OLE` object that represents a COM object based on the specified path; otherwise, it fails and returns `undef`. [Example 5.2](#) demonstrates the `GetObject()` method.

This example attempts to discover the title, subject, and author of all Word documents in a given directory. If line 12 cannot obtain a COM object-based return, the script continues on to the next file. Otherwise, the COM object is queried for its type and is compared with the type that a Word document would report (`Document`). If indeed the query is successful and the COM object is a Word document, `Title`, `Subject`, and `Author` properties are queried and printed.

### **Example 5.2 Using `GetObject()` to generate a summary of all `.DOC` files**

```
01. use Win32::OLE;
02. my $Dir = "c:\\temp";
03. my $FileSpec = "*.doc";
04. foreach my $File ( glob( "$Dir/$FileSpec" ) )
05. {
06.     ProcessFile( $File );
07. }
08.
09. sub ProcessFile
10. {
11.     my( $File ) = @_;
12.     my( $Doc ) = Win32::OLE->GetObject( $File ) || return;
13.     return if( "_Document" ne Win32::OLE-
>QueryObjectType( $Doc ) );
14.     if( $Doc )
15.     {
16.         print "$File:\n";
17.         print "\tTitle: ", GetProperty( $Doc, 'Title' ), "\n";
18.         print "\tSubject: ", GetProperty( $Doc, 'Subject' ), "\n";
19.         print "\tAuthor: ", GetProperty( $Doc, 'Author' ), "\n";
20.         print "\n";
21.     }
22. }
23.
24. sub GetProperty
25. {
26.     my( $Object, $Property ) = @_;
27.     my $Prop = $Object->BuiltInDocumentProperties( $Property );
28.     return( $Prop->{Value} );
29. }
```

### **Tip**

You might be wondering how `Win32` can possibly know what class ID to use when using `Win32::OLE->GetObject()`. After all, you supply only a path to a file; you are not supplying the name of the class. The operating system will try four steps to determine which class ID to use:

1. When `Win32::OLE`'s `GetObject()` method asks the OS for a COM object based on a filename, it will first look through the current list of objects in memory. If an object already exists representing the specified file, the OS returns that COM object.
2. If there are no currently instantiated objects based on the file, the OS next tries to read the class ID from the file. This can occur only if the file is a structured storage file. These types of files are created by programs, such as MS Office 97, in which each file can have a title, subject, author, and other similar information in addition to the files class ID. The DocFile Viewer application (`DFVIEW.EXE`) that comes with Microsoft's development programs (such as Visual C++) enables you to peruse the contents of such files.
3. If the class ID is not yet discovered, the OS resorts to walking the keys found in the Registry hive:

`HKEY_CLASSES_ROOT\FileType`

The interesting thing about these keys is that their name is a class ID. The way that the OS determines the class ID that matches the specified file is by systematically opening each of the key's subkeys. A subkey will hold a data value containing four comma-separated fields such as:

`0,2,FFFF,DBA5`

The format for this data is:

`POSITION, BYTES, MASK, RESULT`

The OS will then open the specified file and load the number of bytes specified by `BYTES` from the location specified by `POSITION`. Next, the loaded data will be logically AND'ed with the value of `MASK`. If the resulting data is the same as `RESULT`, the OS assumes that the file matches this class ID.

4. If the preceding steps were inconclusive, the OS will walk through all the Registry keys that represent file extensions, looking for one that matches the extension of the specified file. When one matches, the class name is extracted from the key's data value, and the class ID is looked up based on that class name.

If all these steps fail, the OS gives up and `Win32::OLE->GetObject()` will fail.

## Retrieving the COM Object Type

After you have a COM object, you might need to discover what type of COM object it is. You can use the `QueryObjectType()` for this:

```
Win32::OLE->QueryObjectType( $Object );
```

The only parameter (`$Object`) is a `Win32::OLE` object. This can be obtained by calls to `new()`, `GetActiveObject()`, `GetObject()`, or some COM object's internal method.

The value returned is based on the context of the call. If the method is called in a scalar context, the class name is returned. If the method is called in an array context, the returned value is an array consisting of the type library (described later) and a class name.

If the `QueryObjectType()` method is successful, it returns type information for the COM object; otherwise, it fails and returns `undef`.

The returning class name is defined by the COM object itself, so there is no way to accurately predict what an object's type might be. It is best to experiment with different COM objects to know what strings they will return.

The script in [Example 5.3](#) accepts an input parameter that is some document file. The script tries to create a COM object based on the file (such as a Word document or an Excel workbook) and prints the type of the object.

#### ***Example 5.3 Determining a COM object's class name***

```
01. use Win32::OLE;
02. my $File = $ARGV[0];
03. if( my $Object = Win32::OLE->GetObject( $File ) )
04. {
05.     my $Type = Win32::OLE->QueryObjectType( $Object );
06.     print "The object type for $File is '$Type'.\n";
07. }
```

[Example 5.4](#) is the same as [Example 5.3](#) except that it also prints out the object's type library.

#### ***Example 5.4 Determining both a COM object's type library and its class name***

```
01. use Win32::OLE;
02. my $File = $ARGV[0];
03. if( my $Object = Win32::OLE->GetObject( $File ) )
04. {
05.     my @TypeInfo = Win32::OLE->QueryObjectType( $Object );
06.     print "The object type for $File is '$TypeInfo[1]'\n";
07.     print "The type library is: '$TypeInfo[0]'.\n";
08. }
```

## **Interacting with COM Objects**

After you have successfully obtained a COM object, your Perl script can begin having fun. You need to familiarize yourself with a couple concepts before all this will make sense: COM objects and COM collections.

## COM Objects

When you create a `Win32::COM` object, you are creating a Perl object that represents a COM object. The COM object can contain properties and methods. By interacting with the properties and methods, you can change the state of the object and command the object to perform tasks.

If you have never dealt with COM objects before, think about starting MS Word. After you have Word running, you can consider the program's main window to be a Word application object. You can interact with it by pressing toolbar buttons and selecting menu items. You select the menu's File, Open command. It enables you to specify a file to open, and after the file is loaded, you now have a document window. You can consider that window to be a document object.

This object enables you to edit text, format, print, and save. These would be some of the document object's methods. You could tell the document that it is read-only, that it is to be saved as a text file, or that the left margin is at one inch. These are considered to be properties of the document, or the document object's properties.

Basically, this is what a COM object is. Almost everything is an object—applications, documents, paragraphs. Even a font is an object because it represents a bunch of properties such as the font type, size, and attributes (like bold and italics). One COM object is a bit different, however: the collection object.

## COM Collections

At times you need to select many objects that are the same type. MS Word might have five documents open, for example. If you wanted to save and close all of them quickly, you could select a document object, tell it to save and close, and then move on to the next document object, repeating this pattern until all the documents are closed. If you have 100 open documents, however, this could take quite some time. You could, instead, request a *collection* of document objects. A collection is a special COM object that represents many of a particular type of object. If you could obtain a COM document collection object that represented all 100 of the open documents, you could tell the object to save and quit in one mighty stroke.

Collection objects are obtained by calling certain methods. If you have a Word COM object, for example, you could call the `Documents()` method:

```
$AllDocuments = $Word->Documents();
```

This would return a COM collection object representing all the documents that Word has opened. Collection objects, like other objects, have properties and methods. One property most all collections have is the `Count` property, which indicates how many objects the collection represents. One method that most collection objects have is the `Add()` method. This method will create a COM object based on the type that the collection represents. If you were to use the following code, for example:

```
$Doc = $Word->Documents()->Add();
```

You would be requesting a new Word document object.

Collection objects can make a job not only easier but faster. Consider [Example 5.5](#) and [Example 5.6](#).

### **Example 5.5 Using a collection object**

```
01. use Win32::OLE;
02. my $Class = "Word.Application";
03. my $Word = Win32::OLE->GetActiveObject( $Class )
04.           || die "Could not find a $Class object.\n";
05. # Save and close all documents
06. my $Docs = $Word->Documents() || die "Can't find collection\n";
07. $Docs->Save();
08. $Docs->Close();
```

### **Example 5.6 An alternative to using collection objects**

```
01. use Win32::OLE;
02. my $Class = "Word.Application";
03. my $Word = Win32::OLE->GetActiveObject( $Class )
04.           || die "Could not find a $Class object.\n";
05. #Get the collection of all word objects
06. my $Docs = $Word->Documents() || die "Can't find collection\n";
07. # Find How many documents are open
08. my $iTotal = $Docs->{Count};
09. # Save and close all documents
10. while( $iTotal )
11. {
12.     my $Doc = $Docs->Item( $iTotal );
13.     $Doc->Save();
14.     $Doc->Close();
15.     $iTotal--;
16. }
```

These two examples both perform the same task. They save and close all open Microsoft Word documents. Because [Example 5.5](#) does not loop and call the `Save()` and `Close()` methods for each document, the overall effect is that it is significantly faster than [Example 5.6](#). This speed increase is achieved because it uses the server application to save and close the documents. The COM server can do this much faster than the controller script, which must call in to the COM server to obtain the next document, save it, and then close it. Each of these calls imposes latency.

Notice that, in [Example 5.6](#), the `while` loop starts by passing the `$iTotal` value into the `Item()` method (line 12). This works because Microsoft Word's document collection does not accept a 0 value to be passed in. When the `while` loop determines that `$iTotal` has been decremented to 0, then it ends.

### **Tip**

*Most methods that are plural (as in `Documents()`, `Worksheets()`, and `Paragraphs()`) return COM collection objects. For example, the code*

```
$AllDocs = $Word->Documents();
```

*returns a COM collection object that represents all the open documents. The following code will save and close all the open documents:*

```
$AllDocs = $Word->Documents();
$AllDocs->Save();
$AllDocs->Close();
```

*These pluralized methods usually will return one COM object if you pass in an object identifier. If you call the `Documents()` method passing in a number, for example, it will return a single COM document object represented by the passed-in value. The following code segment will return the first open Word document:*

```
$Doc = $Word->Documents( 1 );
```

*These methods typically enable you to pass in a string to use as an identifier. The string represents the name of the object you seek:*

```
$Doc = $Word->Documents( "Test.doc" );
```

## Tip

*When accessing objects from a collection object, it is important to know that objects are not always numbered starting with the number 1. An Excel workbook object has a method called `Worksheets()` that returns a collection object of worksheets. To obtain the first worksheet, you would call `Worksheets(1)`, not `Worksheets(0)`. This is important because an attempt to obtain a COM object calling `Worksheets(0)` will fail.*

*However, other collections might accept a value of 0. It is totally up to the application hosting the COM collection.*

## Using COM Objects and Collections

Suppose you have a word-processing document that has 10 paragraphs. In Word, each paragraph is an object. You could say that the document has a collection of 10 paragraph objects. Just as the Word application object can contain a collection of different documents (if you have opened up many documents), so can a document object contain a collection of many paragraph objects. If you were to select half of the paragraphs and change the font size, you would be changing the property of a collection of paragraph objects. This is the same as if you selected half of the paragraphs in your document using your mouse and then, while the paragraphs were highlighted, changed the font size. We, as humans, say that you are just changing the font size for a bunch of paragraphs, but a program would say that it is altering various font properties on a collection of paragraph objects. This is what [Example 5.7](#) illustrates.

### **Example 5.7 Using collections to alter many objects simultaneously**

```
01.    use Win32::OLE;
```

```

02. my $File = "c:\\files\\myfile.doc";
03. my $Doc = Win32::OLE->GetObject( $File ) || die;
04. $Doc->{Application}->{Visible} = 1;
05. my $Count = int( $Doc->Paragraphs()->{Count} / 2 );
06. my $StartingParagraph = $Doc->Paragraphs(1)->{Range}->{Start};
07. my $EndingParagraph = $Doc->Paragraphs($Count)->{Range}->{End};
08. my $Collection = $Doc->Range(
09.     {
10.         Start => $StartingParagraph,
11.         End => $EndingParagraph
12.     } );
13. $Collection->{Font}->{Size} = 20;
14. sleep( 1 );
15. $Collection->{Font}->{Bold} = 1;
16. sleep( 1 );
17. $Collection->{Font}->{Italic} = 1;
18. sleep( 1 );
19. $Doc->Save();
20. sleep( 1 );
21. $Doc->Close();

```

Line 3 of [Example 5.7](#) attempts to load a persistent (saved) Word document object. Line 4 tells the COM object's application (Microsoft Word) to turn visible so you can see what is going to happen. Line 5 then determines how many paragraphs exist by referring to the `Count` property from the paragraph collection object and halves it. The `Paragraphs()` method returns a collection of all the paragraph objects that represents all paragraphs in the document. It contains properties and methods just as a regular object does.

Line 8 calls the `Range()` method from the document. The range is between the beginning of paragraph 1 (the paragraph 1 object is returned by a call to `Paragraphs(1)` in line 6) and the end of the middle paragraph because the example attempts to change the fonts of the first half of the document only. The `Range()` method will return the collection of characters between the specified points. Notice that we are using named parameters (described in the next section) in the call of the `Range()` method.

Lines 13–15 treat the collection as if it were any other object (such as a character or paragraph object), changing the `Bold`, `Italic`, and `Size` properties in the `Font` object (the value of the `Font` property is a `Font` object). Don't mind the calls to the `sleep()` function because it is there to slow down the script. Otherwise, fast machines will process it too quickly to see.

## **COM Object Methods**

A COM object usually has methods that provide some sort of functionality. You cannot call methods for one type of object from another. For example, consider an Excel workbook containing worksheets. The workbook object will have a `Save()` method, but the worksheet will not. Because worksheets are not individually saved, it makes no sense for a worksheet object to be saved. Workbooks, on the other hand, can be saved, and this will save all the workbook's worksheets.

After a Perl script makes changes to a worksheet and it is time to save the changes, a workbook object must be used to call `Save()`. If line 10 of [Example 5.8](#) was changed to:

```
$Worksheet->Save()
```

The script would fail because Excel's worksheet objects do not have a `Save()` method.

In [Example 5.8](#), couple of interesting things are worth pointing out. Line 5 obtains a COM object that represents the Excel application. This is used in lines 6, 7, and 11. Line 6 forces Excel to show itself so that the user can see it. Line 7 forces the workbook's window to become visible. This is done by obtaining the COM object that represents the workbook's display window. Notice that this is done by calling into the application's (`$ExcelApp`) `Windows()` method, passing in the name of the workbook. Finally, the `Quit()` method is called from the application's COM object on Line 11.

### **Example 5.8 Calling the Save() method from an Excel worksheet**

```
01. use Win32::OLE;
02. my $File = shift @ARGV || "c:\\temp\\mytest.xls";
03. my $Workbook = Win32::OLE->GetObject( $File )
04.           || die "Could not load $File.\n";
05. my $ExcelApp = $Workbook->{Application};
06. $ExcelApp->{Visible} = 1;
07. $ExcelApp->Windows( $Workbook->{Name} )->{Visible} = 1;
08. my $Worksheet = $Workbook->Worksheets(1);
09. $Worksheet->Range("A1")->{Value} = "My Unique Change.";
10. $Workbook->Save();
11. $ExcelApp->Quit();
```

### **Named Parameters**

When you call a method and pass in parameters, they are *positional parameters* by default. This means that the parameters depend on their position to indicate their meaning. For example, if a method assumes that the first parameter passed in is a filename and the second parameter is the file type, then this order must be respected; otherwise, the method will most likely fail.

Some methods have a parameter order that needs to be respected, but some of the parameters might be optional. This poses quite a dilemma. Take the `SaveAs()` method in Microsoft Word. All its parameters are optional, but there is an order to these optional parameters. The `SaveAs()` method's first parameter is a file path and the second is the file format, but both of them are optional. If you need to specify the third parameter (password) but do not want to specify the first (file path) or the second parameter (file format), then you have a problem because parameter order must be respected. How do you specify the third parameter without specifying the first two? This is accomplished by using `undef`. In this case, you would use `undef` for the first two parameters, as in:

```
$Doc->SaveAs( undef, undef, "NewPassword" );
```

Some methods allow the use of *named parameters*. These are parameters that are associated with a name. When using named parameters, you must first pass in any required positional parameters, then you can pass in a hash reference with key/value pairs. These pairs constitute the names and

values of the named parameters. It is important to note that some positional parameters might be required before any named parameters are specified. Suppose some method requires two parameters and a bunch of optional named parameters. You would first specify the positional parameters and then the hash array of named parameters:

```
$Object->Method( $RequiredParam1, $RequiredParam2,
{ Named1=>$NameParam1,
Named2=>$NameParam2 } );
```

For example, a Word document's `SaveAs()` method enables you to supply optional parameters such as file format and file name. [Example 5.9](#) shows this in action.

### **Example 5.9 Using named parameters in a method call**

```
01. use Win32::OLE;
02. use Win32::OLE::Const "Microsoft Word";
03. my $File = "c:\\temp\\myTest.doc";
04. my $NewFile = "c:\\temp\\myTest.rtf";
05. my $Doc = Win32::OLE->GetObject( $File ) || die;
06. my $App = $Doc->{Application};
07. $App->{Visible} = 1;
08. $App->{Options}->{SavePropertiesPrompt} = 0;
09. $Doc->SaveAs(
10.   {
11.     Filename => $NewFile,
12.     FileFormat => wdFormatRTF,
13.   } );
14. $Doc->Close();
```

[Example 5.10](#) was designed to work with Word 9.0 (Office 2000), so line 8 might not be applicable on older versions of Word. (This line prevents Word from requesting that you enter document information before the save occurs.)

Any number of named parameters can be passed into a method call, but they must all reflect some meaning to the method. Supplying named parameters that are not recognized by the method might cause the method to fail.

#### **Tip**

The `Win32::OLE::Const` extension is an important and useful extension, but it is not optimized (as of this writing). Therefore, you might find multiple warning messages are printed if you specify the `-w` Perl flag.

### **The Invoke Method**

Under certain circumstances, it might not be possible to access a method or property because of its spelling. Consider a non-English language version of Win32 that uses dialectic marks such as the umlaut (as in ö), a ç character, or a n~ character. These types of nonstandard ASCII characters are

not supported by Perl. Therefore, it is very difficult to access a method whose name consists of such characters. For this reason, the `Invoke()` method has been exposed for a Perl script:

```
$Object->Invoke( $Method [ , @Parameters ]);
```

The first parameter (`$Method`) is a string that represents either a method or property.

The second parameter (`@Parameters`) is actually a list of parameters, as many as are needed for the specified method. If the first parameter represents a property, this method will only return the property's value—it will not set the property. Therefore, only the first parameter is required when dealing with a property.

Just like any other COM object method, you can supply a reference to a hash (or an anonymous hash) to specify named parameters. See the previous section in this chapter that discusses named parameters for more information.

**Example 5.10** makes use of the `Invoke()` method. In this example, `Invoke()` is used three times: once in line 4 and twice in line 11. Line 4 could have been written as:

```
my $iTotal = $App->Documents()->{Count};
```

The line was written the way it was to illustrate that any method or property can be referenced with a call to `Invoke()`. In the case of line 4, `Invoke()` is called passing in the name "`Documents`". Perl will locate the method or property (in this case, the method) by the specified name and execute it.

Line 11 is fairly interesting because it calls the `Documents()` method using `Invoke()`, as it did in line 4. However, it also passes in a property, `$iCount`. This is the equivalent of calling `Documents($iCount)`. The result of this call to `Invoke()` is a COM object representing a Microsoft Word document. Line 11 immediately then calls `Invoke()` again from the returned Word COM object. This time it passes in the "`Name`" value. The result is that the `Name` property value is returned.

### **Example 5.10 Using the `Invoke()` method**

```
01. use Win32::OLE;
02. my $App = Win32::OLE->GetActiveObject( "Word.Application" )
03.       || die "Word is not running.\n";
04. my $iTotal = $App->Invoke( "Documents" )->{Count};
05. my $iCount = 0;
06. print "There are currently $iTotal open document(s):\n";
07. while( $iCount++ < $iTotal )
08. {
09.     printf( "\t%02d) %s\n",
10.             $iCount,
11.             $App->Invoke( "Documents", $iCount )-
>Invoke( "Name" ) );
12. }
```

## **Object Properties**

Properties are simple variables that describe the state of the object. If you have a Word document object, for example, it will have a "Name" property that represents the name of the document. Additionally, it will have a "FullPath" property that describes the location where the file resides (either as a local file or a UNC).

Properties can be write-protected (that is, they are read-only). Usually properties that reflect a condition of the object are read-only because changing the value would not change the condition. Word has several protected properties, for example, that describe how many characters, words, and paragraphs exist in a document object, for example.

A COM object's property is accessed as if it were a hash key in the object. A Word document's name, for example, can be discovered, such as:

```
$DocumentName = $WordObject->{Name};
```

Setting a property value is as you would expect for any hash reference:

```
$WordObject->{Name} = "foo.doc";
```

Interestingly enough, some properties are writeable but are overwritten by other actions. You can change a Word document object's `Name` property, for example, but when you save the document, the `Name` property is changed back to its preceding value. (The correct way to change the name is to call the `SaveAs()` method.)

## Tip

`Win32::OLE` does not require your Perl scripts to respect a COM object's properties and methods case—that is, they are not case sensitive. The following two calls are identical:

```
$Object->SaveAs( $File );
$Object->saveas( $File );
```

And the following property accesses are identical:

```
$Object->{Value} = 1;
$Object->{vALuE} = 1;
```

Additionally, if a method call does not have any parameters passed in, the parenthesis can be left off. Therefore the following are identical:

```
$Object->Document()->{Value};
$Object->Document->{Value};
```

Because different programmers have different styles, it can be quite confusing whether the following line is accessing a method or a property (because it could be either):

```
$Object->Count;
```

To make your code understandable by other programmers, you might want to follow these conventions:

- Use the case that any documentation suggests. MS Word's documentation, for example, indicates that the `SaveAs()` method has a capital S and A. You should respect this case.
- All methods should always use parentheses, even if no parameters are passed in.
- All properties should be enclosed in braces, just as if they were hash keys.

Some property values are really COM objects. An Excel worksheet contains a property called `Parent`, for example. The value of this property is a COM object, which represents the workbook in which the worksheet resides (the worksheet's parent object). Likewise, that workbook object has a `Parent` property, which is a COM object that represents the Excel application. [Example 5.11](#) illustrates this by creating a simple Word document and using its `Parent` property to access the document object's parent object (which happens to be the Word application).

### **Example 5.11 Accessing COM objects from within COM objects**

```
01. use Win32::OLE;
02. my $Doc = new Win32::OLE( "Word.Document" );
03. print "Document name is ", $Doc->{Name}, "\n";
04. print "Application name is ", $Doc->{Parent}->{Name}, "\n";
05. $Doc->Close();
```

If you are querying a property, you do not necessarily need to enclose it in braces. For example

```
print $Object->Parent->Name;
```

is the same as

```
print $Object->{Parent}->{Name};
```

However, if you are setting a property value, you must enclose it in braces. If a `Win32::OLE` cannot resolve a method, it will try to access it as a property. This is good to know because many programmers write code using this feature, which could mislead you to believe that `Parent` is a method and not a property.

[Example 5.12](#) demonstrates how a Perl script can use COM to interact with two different applications. In this example, Microsoft Word will open a document and transfer its text into a new Excel spreadsheet.

### **Example 5.12 Moving text from a Word document to an Excel Worksheet**

```
01. use Win32::OLE;
02. my $WordFile = "c:\\temp\\mytest.doc";
03. my $ExcelFile = "c:\\temp\\mytest.xls";
```

```

04. my $Doc = Win32::OLE->GetObject( $WordFile ) || die;
05. my $ExcelApp = Win32::OLE->GetObject( "Excel.Application" );
06. if( ! $ExcelApp )
07. {
08.     $ExcelApp = new Win32::OLE( "Excel.Application", "Exit" ) || die;
09. }
10. my $WorkBook = $ExcelApp->Workbooks()->Add() || die;
11. my $WorkSheet = $WorkBook->Worksheets()->Add() || die;
12. my $WordApp = $Doc->{Application};
13. $WordApp->{Visible} = 1;
14. $ExcelApp->{Visible} = 1;
15. $WorkBook->{Title} = "MyTest";
16. $WorkSheet->{Name} = "MyTest WorkSheet";
17. $ExcelApp->Windows( $WorkBook->{Name} )->{Visible} = 1;
18. my $Paragraphs = $Doc->Paragraphs();
19. my $iTotal = $Paragraphs->{Count};
20. my $iIndex = 0;
21. while( ++$iIndex <= $iTotal )
22. {
23.     my $Paragraph = $Paragraphs->Item( $iIndex )->Range();
24.     my $Text = $Paragraph->{Text};
25.     my $Style = $Paragraph->{Style}->{NameLocal};
26.     my $Column = ( $Style =~ /Heading/i )? "A" : "B";
27.     my $Size = length( $Text );
28.     my $Percent = int( 100 * $iIndex / $iTotal );
29.     print "\rProcessing: $Percent%", " " x 30;
30.     $WorkSheet->Range( "$Column$iIndex" )->{Value} = $Text;
31. }
32. $WorkBook->SaveAs( $ExcelFile );

```

### ***Chaining Methods and Properties***

Just like any Perl function, you can chain method calls and properties together. [Example 5.13](#) illustrates how to do this.

#### ***Example 5.13 Chaining methods and properties***

```

01. use Win32::OLE;
02. my $File = shift @ARGV || "c:\\temp\\mytest.xls";
03. my $Workbook = Win32::OLE->GetObject( $File )
04.           || die "Could not load $File.\n";
05. $Workbook->{Application}->{Visible} = 1;
06. $Workbook->{Application}->Windows( $Workbook->{Name} )-
>{Visible} = 1;
07. # Get the total number of worksheets in this workbook
08. my $iTotal = $Workbook->Worksheets()->{Count};
09. # For each worksheet autofit each column
10. while( $iTotal )
11. {
12.     $Workbook->Worksheets($iTotal)->Columns( )->AutoFit();
13.     $iTotal--;

```

```
14. }
15. $Workbook->Save();
16. $Workbook->{Parent}->Quit();
```

Lines 5 and 6 simply force the application and the workbook window to be visible so you can see what is happening. Line 8 accesses the `Count` property for a collection that is returned by the `Worksheets()` method. This value indicates how many worksheets the Excel workbook contains. The line could have been broken into two parts:

```
$Collection = $Workbook->Worksheets();
$iTotal = $Collection->{Count};
```

By chaining the method call and property access together, it is done in one line and prevents the mess of adding additional variables (in this case, the `$Collection` variable).

A better example of chaining is line 12. This line first calls the `$Workbook` object's `Worksheets()` method, which returns a worksheet COM object that represents a particular worksheet in the workbook. Then the `Columns()` method is called from within the worksheet object. Because no particular column is specified, it returns a collection of all the spreadsheet's columns. This collection is yet another COM object (a collection of `Column` objects). Finally, the `AutoFit()` method is called from within the `Columns` collection object. The overall effect of all this is that all the columns in the specified worksheet are autofitted. In other words, the columns are sized so that you can see their contents. If you broke this out it could take several lines, but there is no need to do so because you can chain the properties and methods together.

## Note

*One of the most discouraging aspects of programming Perl scripts that access COM objects is that there are no standards for properties and methods. That is, if you are trying to automate Excel and Photoshop, the two could have radically different property names and methods. Additionally, methods that are named the same could take different parameters.*

*If you have an Excel COM object and want to print the name of the application, for example, you could use:*

```
print $Object->{Name};
```

*But Photoshop would require:*

```
print $Object->{FullName};
```

*This can be quite confusing for someone who is just learning how to interact with a particular object. Some objects expose an `Open()` method, for example, whereas others might use a `Load()` method.*

*The point here is that it would be futile to make a script that attempts to treat every object the same. It would be best to use the `Win32::OLE->QueryObjectType()` method to determine what kind of object you have and interact with it accordingly.*

## Tip

*A COM object can expose both methods and properties. Properties can be either set or queried (or in the case of a read-only property, it can only be queried). Methods can only be called, optionally passing in parameters.*

*Now for the bold truth ... there is no difference between properties and methods. Deep inside an application's object model, any time a script accesses a property, the object model will call an internal method. For example, let's say you are querying a `Count` property, as in:*

```
print $Object->{Count};
```

*Internally, the COM server application would call a method with the name of `getCount()`. If your script is setting the property, as in:*

```
$Object->{Count} = 33;
```

*Internally, the COM server application would call a method with the name of `putCount( 33 )`.*

*If you call a normal method, as in:*

```
$Object->Save( "c:\\Foo.txt" );
```

*Internally, the COM server application would call a method with the name of `Save( "c:\\foo.txt" )`.*

*This is the reason why you can leave parentheses off a method call such as:*

```
$Object->Documents->Count;
```

*In this case, you can't tell if `Documents` or `Count` are methods, properties, or a mix. Hint: It doesn't matter, because they, internally, are both methods anyway.*

*The only time it does make a difference is if you are setting a property value or passing parameters into a method. However, you use the `Invoke()` method to call a method or query or to set a property.*

# Destroying COM Objects

After you have finished using a `Win32::OLE` object, it needs to be destroyed. Destruction is accomplished in two ways: either by letting the object fall out of scope or by forcing it to destroy itself. Falling out of scope is quite easy; when the function, code block, or script terminates, the object is destroyed. Forcing the object to destroy itself requires that you call the object's `DESTROY()` method:

```
$Object->DESTROY();
```

This method takes no parameters and causes the object to terminate. Generally speaking, this method is used internally by the object itself and not by a Perl script.

It is possible that a script could run for hours but only need to create a COM object for just a few minutes. Assuming that for some reason you need to release the object, you could call the `DESTROY()` method.

## OLE Errors

The `Win32::OLE` extension supplies a method to query any error that the preceding action on a COM object might have produced. Every interaction with a COM object (a call into a method or a query or setting of a property) will set the error state, even if no error was generated. Therefore, you must always query the error state before any other interaction with any COM object.

The error state is obtained with a call into the `LastError()` method:

```
Win32::OLE->LastError();
```

The `LastError()` method will return a value based on context. If the method is called in a numeric context, the returned value is the reported error number. If the method is called in a string context, the return value is the error text message.

This method always returns the current error state that was last generated by a `Win32::OLE` object.

A return value of `0` indicates a successful nonerror state.

[Example 5.14](#) shows how the `LastError()` method returns two different values based on context. Line 5 forces the return context to be numeric, so result that is the error code is in a numeric value. Line 6 will result in a string describing the error.

### **Example 5.14 Examining the `LastError()`**

```
01. use Win32::OLE;
02. my $Object = new Win32::OLE( "Some.Unknown.Object" );
03. if( "Win32::OLE" ne ref $$Object )
04. {
05.     my $NumericValue = 0 + Win32::OLE->LastError();
06.     my $StringValue = Win32::OLE->LastError();
07.     print "Numeric value: $NumericValue\n";
```

```
08. print "String value: $StringValue\n";
09. }
```

## Miscellaneous OLE Items

A few additional `Win32::OLE` surprises await your use. These are the lesser-known aspects, but they can be very powerful in your programming arsenal. Some of these things are taken from Visual Basic, but they do come in very handy in Perl scripts.

### Events

Windows applications often produce internal messages that are known as *events*. An event is an indication that something has occurred. You can imagine a simple clock program that has an alarm set for 7:30 in the morning. When the program notices that the time is 7:30, it could fire off an event. Any application that is listening for this event would be notified and can react. For example, let's say you have two applications waiting for the alarm event. One will start brewing a pot of coffee when it notices that the event has occurred. Another one turns on the radio to wake you up.

In another example, you could have an event fire every time an application notices that a server has gone down. Some applications might wait for that event so that they can page the system administrator.

Perl has the capability to listen to events. To do this, however, we need to describe in a few words how events work. Normally, a Windows application has what is known as a *message loop*. The application will start, initialize itself, create any required objects and windows, and then enter its message loop. This loop will continue until the application terminates. All interaction that occurs in the program (such as the user selecting buttons, moving around the window, or the application waking up every few seconds to update some display) results due to a message being posted onto its *message queue*.

For example, when you click on a button of some application's window, the operating system posts a message onto the application's message queue. This message indicates the coordinates of the mouse cursor, which button was selected, and other information. As the application checks its message queue, it will see this message and process it. This processing might cause the application to create new windows, process data, update a database, or even send data over the Internet. After it has finished doing what it is doing, the application loops back and waits for any new messages. This continues throughout the lifetime of the application and is known as the application's message loop—because the application is really running one big loop that continuously checks for incoming messages.

An application's message loop might respond to certain messages by posting messages to other applications. This is how events are fired. For example, if your Perl script subscribes to some application's event, it will have to create a message queue to accept any incoming messages. The application whose events you are monitoring can post event messages to the Perl script's message queue. The script must be in a message loop to check the messages, and when it finds one, it will call some Perl function. The Perl function then processes the event (for example, it sends out a page to the system administrator).

So this all seems a bit complex, but it really is pretty simple to set up. First you need to load the `Win32::OLE` extension importing the "Events" option:

```
use Win32::OLE 'EVENTS';
```

This will force `Win32::OLE` to prepare itself for listening for events.

Next you need to create a COM object, which you are interested in having monitor events. Let's try creating a new Internet Explorer window:

```
$Ie = new Win32::OLE( "InternetExplorer.Application" );
```

We are almost finished! Next you simply need to start listening for events. This is done with the `WithEvents()` function:

```
Win32::OLE->WithEvents( $COMObject [, $EventHandler [, $Class ] ] );
```

The first parameter (`$COMObject`) is a COM object. By default, `Win32::OLE` will try to determine the default event source for the particular COM object. However, this is not always possible (because either there is no default event source or the COM object does not support what is known as the `IProvideClassInfo2` interface). If this is the case, you will need to specify the optional third parameter.

The optional second parameter (`$EventHandler`) can be either a code reference or the name of a Perl package. If a code reference is specified, then any event that occurs will call into the specified code reference. When this happens, the first two parameters passed into the code reference will be (in order) the COM object (the first parameter passed into `WithEvents()`) and a string containing the name of the event. Any additional parameters passed in are event specific.

The second parameter might also be the name of a Perl package. In this case, the package will need to have a subroutine for each event that it wants to catch. For example, if a `CloseWindow` event occurs, then the package's `CloseWindow()` subroutine will be called. The first parameter passed into the subroutine will be the COM object, as previously noted, followed by any number of event-specific parameters. If a subroutine does not exist for a given event, the event will not occur.

The optional third parameter (`$Class`) indicates that you want to receive events for the specified class. Normally this is not needed, and Perl will assume it should use the default event class—assuming one exists. By specifying this parameter, you are forcing Perl to listen for the events defined by a particular class. The only way to know if you should use this is by either experimentation or reading documentation.

If the `WithEvents()` call fails, then `Win32::OLE->LastError()` will report a non 0 value.

You can register for as many events from as many COM objects as needed. To deregister and stop receiving events, just call the `WithEvents()` function passing in only the first parameter.

After you have registered to receive events, you must periodically check the message queue and process any pending messages. There are two ways of creating message loop. In the first, the `SpinMessageLoop()` function comes in:

```
Win32::OLE->SpinMessageLoop();
```

This function processes all pending messages from the message queue and does not return any value. Normally, a Perl script does not need to call this function because it is automatically called every time the Perl script calls a COM method or accesses a COM object's property.

Because `SpinMessageLoop()` returns after the message queue has been processed, your script will need to create a loop in which `SpinMessageLoop()` is repeatedly called. It is a good idea to put a slight delay in the loop to prevent your script from sucking up CPU time needlessly. A delay of 50 milliseconds typically is fine. You can make such a small delay by calling `Win32::Sleep( 50 );`.

However, there is a second, more efficient way to emulate a message loop: using the `MessageLoop()` function:

```
Win32::OLE->MessageLoop();
```

The function will continuously process the message loop. When the message queue is empty, the function will wait for more messages. The operating system will prevent the function from unnecessarily using CPU time when it does not need to.

After `MessageLoop()` has been called, the only real way to cancel and further process a Perl script is to post a quit message to the application's message queue. This can only be done by the user closing a window, another application, or calling the `QuitMessageLoop()` function:

```
Win32::OLE->QuitMessageLoop();
```

This would have to be called from an event function. After it is called, the message loop will see the quit message and terminate the call to `MessageLoop()`.

## Note

*The `MessageLoop()` and `QuitMessageLoop()` functions were introduced in Perl 5.6. Therefore, older versions of Perl might not have access to these functions. Instead, you should call `SpinMessageLoop()` from within a loop (such as a do/while loop).*

[Example 5.15](#) demonstrates using events. Line 2 requests a new Internet Explorer window. Line 3 registers our `EvenTRoutine` function that is called when IE's `DwebBrowserEvents2` class fires an event. Lines 4 through 7 check to see if the event registration was successful. Line 8 simply forces the browser's window to become visible.

Lines 9 through 13 emulate a message loop. If the script is running on Perl v5.6 or higher, this block of code could be replaced with a single call to `MessageLoop()`.

Lines 15 through 55 define the event handler subroutine. If any of the events the script registered for are fired, a call to this subroutine will be made. The first parameter passed in `Eventroutine` is

the COM object being monitored (the IE browser object). The second parameter passed in will be a string identifying what event has occurred. The remaining parameters passed in are event specific.

This particular example code prints out a page's cookies whenever the page has been downloaded. Additionally, it prevents any new windows from being spawned; no more annoying windows will pop open from mischievous Web sites.

### **Example 5.15 Using events with Internet Explorer**

```
01. use Win32::OLE 'EVENTS';
02. my $IE = Win32::OLE->new( "InternetExplorer.Application",
03. "Quit" ) || die;
04. Win32::OLE->WithEvents( $IE, \&EventRoutine,
05. "DWebBrowserEvents2" );
06. if( 0 != Win32::OLE->LastError() )
07. {
08.     die scalar Win32::OLE->LastError();
09. }
10. $IE->{Visible} = 1;
11. while( 1 )
12. {
13.     Win32::Sleep( 50 );
14.     Win32::OLE->SpinMessageLoop();
15. }
16. sub EventRoutine
17. {
18.     my( $Obj, $Event, @Args ) = @_;
19.     if( "DocumentComplete" eq $$Event )
20.     {
21.         my $Url = $Args[1]->Value();
22.         print "Document complete\n";
23.         # ONLY display info if $Url is not "". Otherwise the
24.         # document
25.         # download may have been canceled so there is nothing to
26.         # display.
27.         if( ( "" ne $$Url ) && ( "" ne $$Obj->{Document}-
28. >{Cookie} ) )
29.         {
30.             my @Values = split( /;\s*/ , $Obj->{Document}-
31. >{Cookie} );
32.             my $Cookie;
33.             print " URL:: $Url\n";
34.             foreach my $Cookie ( @Values )
35.             {
36.                 $Cookie =~ s/%(\w\w)/pack( "c", hex( $1 ) )/gei;
37.                 print " Cookie:: $Cookie\n";
38.             }
39.             print "\n";
40.         }
41.     }
42.     elsif( "NewWindow2" eq $$Event )
43.     {
44.         $IE->Quit();
45.     }
46. }
```

```

39.     my( $Window, $CancelVariant ) = @Args;
40.     print "A new browser window is requested. Cancelling the
request!\n";
41.     # Don't allow any new child windows
42.     $CancelVariant->Put( 1 );
43. }
44. elsif( "NavigateComplete2" eq $$Event )
45. {
46.     my( $HtmlDoc ) = $Args[0]->{Document};
47.     print "Navigation complete: ", $Args[1]->Value(), "\n";
48.     print " Cookie:: ", $HtmlDoc->{Cookie}, "\n";
49. }
50. elsif( "OnQuit" eq $$Event )
51. {
52.     exit();
53. }
54. return;
55. }

```

## The `in()` Function

When you have a COM collection object, you really have a special object that, for all practical purposes, is similar to a Perl array. The problem is that the object you are working with is a `Win32::OLE` object, not an array.

If you were to obtain a collection object of open workbooks from Excel, you might want to print the name of each workbook. The problem is that you cannot access the collection object as if it were an array, so you have to walk through a loop accessing each workbook name one at a time, as in [Example 5.16](#)

### ***Example 5.16 Accessing items in a collection—the hard way***

```

01. use Win32::OLE;
02. my $Excel = Win32::OLE->GetActiveObject( "Excel.Application" )
03.             || die "Excel is not running.\n";
04. my $Workbooks = $Excel->Workbooks();
05. my $iTotal = $Workbooks->{Count};
06. my $iIndex = 0;
07. while( $iIndex++ < $iTotal )
08. {
09.     print "Workbook $iIndex: ", $Workbooks->Item($iIndex)-
>{Name}, "\n";
10. }

```

This is where the `in()` function comes in. This will return an array of every element in the collection:

```
@Array = in( $Object );
```

The only parameter passed in is the `Win32::OLE` object, which represents a COM collection object.

The array will consist of the value of the default property of the object, which is usually COM objects (or more precisely `Win32::OLE` objects).

If the `in()` function is successful, it will return an array of values or `Win32::OLE` objects; otherwise, if it fails, it will return nothing. [Example 5.17](#) shows how simple it is to use the `in()` function.

### **Example 5.17 Accessing items in a collection—the easy way using the `in()` function**

```
01. use Win32::OLE 'in';
02. my $Excel = Win32::OLE->GetActiveObject( "Excel.Application" )
03.           || die "Excel is not running.\n";
04. my $iCount = 0;
05. foreach my $WorkBook ( in( $Excel->Workbooks() ) )
06. {
07.   print "Workbook ", ++$iCount, " : $WorkBook->{Name}\n";
08. }
```

It is very important to know that the `in()` method is not exported by default; therefore, you must purposely specify that you want `in()` exported when you load the `Win32::OLE` extension. You do this by specifying the string 'in' with the `use` command:

```
use Win32::OLE 'in';
```

This will import the `in()` function into your main namespace, and you can use `in()` as if it was a normal Perl function. If you do not import it (maybe to prevent a conflict with an already existing `in()` function your script has defined), then you must access it as if it were a method of `Win32::OLE`, as in [Example 5.18](#). Notice how line 5 explicitly calls the `in()` function by specifying its namespace: `Win32::OLE::in()`.

### **Example 5.18 Using `in()` without importing it**

```
01. use Win32::OLE;
02. my $Excel = Win32::OLE->GetActiveObject( "Excel.Application" )
03.           || die "Excel is not running.\n";
04. my $iCount = 0;
05. foreach my $WorkBook ( Win32::OLE::in( $Excel->Workbooks() ) )
06. {
07.   print "Workbook ", ++$iCount, " : $WorkBook->{Name}\n";
08. }
```

## **The `valof()` Method**

When dealing with `Win32::OLE` objects, things can get tricky because the object is really a reference to a blessed Perl object. If you assign the value of a `Win32::OLE` object to a variable, you are really only assigning a reference to an object. Think about it this way: A friend comes to you holding a helium balloon. You ask to hold it, so she ties another string to the balloon and hands the free end of that string to you. Now you are both holding a string that connects to the same balloon.

This is what happens when you run the code in [Example 5.19](#). You have two variables, `$PointToDoc` and `$PointToDoc2`. They are both pointers to the same `Win32::OLE` object. Line 7 causes the object to be destroyed, so by the time line 11 is executed, the object that `$PointToDoc2` points to no longer exists. Therefore, the code will not be able to determine what kind of object it was. You can expect an error to occur in line 11.

Just like the balloon analogy, if the balloon is popped by a pin, it does not matter that two people were holding its strings (`$PointToDoc` and `$PointToDoc2`) because the balloon (the COM object) no longer exists.

### **Example 5.19 Copying a `Win32::OLE` object**

```
01. use Win32::OLE;
02. my $File = shift @ARGV || "c:\\temp\\test.doc";
03. my $PointToDoc = Win32::OLE->GetObject( $File )
04.           || die "Could not open $File.\n";
05. my $PointToDoc2 = $PointToDoc;
06. print "The name of the document is $PointToDoc->{Name}.\n";
07. print "The name of the document is $PointToDoc2->{Name}.\n";
08. $PointToDoc->DESTROY();
09. print "Type of object is ";
10. print "(Next line will result in an error):\n\n";
11. print scalar Win32::OLE->QueryObjectType( $PointToDoc2 ) ;
```

The `valof()` method will return the *default* value of an object. This way, you can obtain the value of an object rather than a reference to it:

```
$Value = valof( $Object );
```

The only parameter is a `Win32::OLE` object. This must be an object or a collection; it cannot be a property.

If successful, the `valof()` method will return the default value for the object; otherwise, it fails and returns `undef`.

The default value for a given object is defined by the object itself. Sometimes it is obvious, like for a Word document the default value is the `name` of the document. You could just as easily access the `Name` property, but different objects have different default values.

This is typically used when you need to access a default value of an object, you have no idea what that object will be, and hence you cannot just access a given property.

The code in [Example 5.20](#) demonstrates the difference between the "value of" a `Win32::OLE` object and the actual object itself.

### **Example 5.20 Using the `valof()` method**

```
01. use Win32::OLE 'valof';
02. my $File = shift @ARGV || "c:\\temp\\test.doc";
03. my $Object = Win32::OLE->GetObject( $File )
04.           || die "Could not open $File.\n";
```

```

05. my $PointerToObject = $Object;
06. my $ValueOfObject = valof( $Object );
07. print "The pointer to the COM object is: $PointerToObject.\n";
08. print "The default value of the COM object is:
$ValueOfObject.\n";

```

The variable `$PointerToObject` is an exact copy of `$Object`. If `$Object` is the string on the balloon, `$PointerToObject` is the second string tied to the balloon. This would make `$ValueOfObject`, let's say the writing on the balloon (like "Eat at Joe's"), the default value of the object (in this analogy, the balloon).

Just as with the `in()` method, `valof()` is not exported by default, so you have to explicitly tell `Win32::OLE` to export it by passing in the string `valof` while loading the extension (refer to line 1 of [Example 5.20](#)

## The `with()` Method

One of the most powerful of the miscellaneous methods that `Win32::OLE` has is the `with()` method. This method was inspired from Visual Basic, where there is a good need for it. In Perl, the same need arises. Basically, it makes tedious tasks a bit easier. If you have an object that you need to set several properties for, it could take several lines of code to do it. Take, for example, a font object. This type of object can contain a font name, size, color, and whether it is bold, italic, or underlined. If you were to write code to set all these values, it would look like [Example 5.21](#).

### ***Example 5.21 Altering properties manually***

```

01. use Win32::OLE;
02. my $Workbook = Win32::OLE->GetObject( "c:\\temp\\mytest.xls" )
03.           || die "Could not open the workbook.\n";
04. $Workbook->{Application}->{Visible} = 1;
05. $Workbook->{Application}->Windows( $Workbook->{Name} )-
>{Visible} = 1;
06. my $Sheet = $Workbook->Sheets( 1 );
07. my $Font = $Sheet->Range("A1:C5")->{Font};
08. $Font->{Name} = "Courier New";
09. $Font->{Size} = 20;
10. $Font->{ColorIndex} = 6;
11. $Font->{Bold} = 1;
12. $Font->{Italic} = 0;
13. $Workbook->Save();
14. $Workbook->Close();

```

The code in [Example 5.21](#) not only is messy, it makes many more calls into the object than it needs to, slowing down your script. You could replace this by using one call to the `with()` method:

```
with( $Object, Property1=>Value1 [, Property2=>Value2
[, ... ] ] );
```

The first parameter is a `Win32::OLE` object.

The second parameter is really a list of key/value pairs. These pairs are the same as you would use when defining a hash. The key is the object's property name, and the value is the value to which you want to set the property. You can have any number of these key/value pairs.

If the `with()` method is successful, all the listed object's properties will have been set to their corresponding values. This method does not return any values.

`with()` is not exported by default, so your script will have to explicitly export it by passing in the value "with" when loading the extension. Notice how this is illustrated in line 1 of [Example 5.22](#).

[Example 5.22](#) requires an explanation for lines 4 and 5. Several versions of Excel have a bug that requires you to make the workbook visible before saving. This is identified by Microsoft's Knowledge Base article Q111247. Line 4 sets the workbook's application (the Excel program) `Visible` property to 1 (`trUE`—sets the object to be visible). Then line 5 tells the same application to locate the window that represents the workbook (by passing in the workbook's name). That window is then set as visible.

### **Example 5.22 Altering properties using the `with()` method**

```
01. use Win32::OLE 'with';
02. my $Workbook = Win32::OLE->GetObject( "c:\\\\temp\\\\mytest.xls" );
03.           || die "Could not open the workbook.\n";
04. $Workbook->{Application}->{Visible} = 1;
05. $Workbook->{Application}->Windows( $Workbook->{Name} )-
>{Visible} = 1;
06. my $Sheet = $Workbook->Sheets( 1 );
07. with( $Sheet->Range("A1:C5")->{Font},
08.       Name => "Courier New",
09.       Size => 20,
10.      ColorIndex => 6,
11.      Bold => 1,
12.      Italic => 013.
13. );
14. $Workbook->Save();
15. $Workbook->Close();
```

For an example of a more administrative nature, see [Example 5.23](#). This code will connect to an IIS web server (either to "localhost" or to one specified as the first parameter in the command line) and add a new virtual directory to the root (the default) web site on that server.

### **Example 5.23 Creating a virtual directory in the default IIS web site**

```
01. # Details on IIS COM objects are found in the SDK
02. # which comes with the IIS server.
03. # This script will create a new "virtual" subdirectory
04. # for the specified web site "Root"
05. # Details on IIS COM objects are found in the SDK
06. use Win32::OLE;
07. $Server = shift @ARGV || "localhost";
08. %WebSite = (
09.   name => "Root",
10.   vdir => "PerlTest",
```

```

11.    path => "c:\\temp",
12. );
13. # Get an IIS COM object for the local machine
14. my $WebService = Win32::OLE->GetObject( "IIS://$Server/w3svc" )
15.           || die Error( "IIsWebServer" );
16. # Get a web server object
17. my $WebServer = $WebService->GetObject( "IIIsWebServer", 1 )
18.           || die Error( "IIIsWebServer" );
19. if( $WebServer->Class() ne "IIIsWebServer" )
20. {
21.   die Error( "IIIsWebServer" );
22. }
23. # Get a virtual web site (the default specified site).
24. my $VRoot = $WebServer->GetObject( "IIIsWebVirtualDir",
$WebSite{name} )
25.           || die Error( "IIIsWebVirtualDir" );
26. # Create a new virtual directory in the site
27. my $VDir = $VRoot->Create( "IIIsWebVirtualDir", $WebSite{vdir} )
28.           || die Error( "IIIsWebVirtualDir" );
29. # Set the new directory path and enable users to read its
files
30. $VDir->{AccessRead} = 1;
31. $VDir->{Path} = $WebSite{path};
32. # Commit our changes
33. $VDir->SetInfo();
34.
35. sub Error
36. {
37.   my( $Object ) = @_;
38.   print "Unable to create a '$Object' object..\n";
39.   print "Error: " . Win32::OLE->LastError() . "\n";
40. }

```

## Variant Data Types

Another important thing to know when using `Win32::OLE` is the concept of a *variant*. The way that `Win32::OLE` actually works is by passing every COM object interaction (either accessing a method or querying or setting a property) through a single method known as `Dispatch()`. This method is the only way in which a Perl `Win32::OLE` object ever actually communicates with a COM object. The reasons for this are rather technical and just a tad beyond the scope of this book. Suffice it to say that every interaction made with a COM object ends up calling the `Dispatch()` method.

The `Dispatch()` method is similar to the `Invoke()` method in that the method or property name is passed in along with any parameters. It is these parameters that we are interested in right now. You see, if you pass a string into a COM object's method, the string must be converted from a Perl string into a string the OLE can understand (a Unicode, length-prefixed, null-terminated string). Likewise, if the method returns a string, it must be converted from an OLE-approved Unicode string into a Perl string.

Now, keeping in mind that this type of data transformation must take place, how could OLE possibly know that we are passing in a string and not binary data or an integer for that matter?

Additionally, how would Perl know that the method is returning a string and not binary data or an array of floating-point numbers? It is because of this exact reason that the *variant* data type was created.

Simply stated, a variant is a data structure designed to accommodate almost any type of variable (strings, nulls, integers, real numbers, dates, Booleans, and so on). When any program (Perl, Visual Basic, C++, whatever) wants to pass data into a COM object using the COM object's `Invoke()` method (what is technically known as the `IDispatch` interface's `Invoke()` method), it must fill out a variant data structure for each parameter that is to be passed in. When the results of the method are received, the data must be extracted out of the data structure as well.

`Win32::OLE` does a remarkably wonderful job at handling this for you automatically. It correctly maps your input and output data to variants invisibly. There are, however, some times when you need to help the extension out a bit. Suppose, for example, you have a variable, `$Cost`, that represents the cost of some product. Assume also that `$Cost` has a value of `35`. When your script passes in `$Cost` as a parameter to a COM object's method, `Win32::OLE` will most likely convert it into an integer form of a variant. This is a problem if the method expects to receive a floating-point value of `35.00`.

To get around this problem, you could create a floating-point (a real data type) variant and pass it in rather than the value of your variable. This means that the `Win32::OLE` extension will not have to convert that particular parameter because it is already in a variant form.

Creating and manipulating variants is accomplished by using the `Win32::OLE::Variant` extension. This comes with the `Win32::OLE` extension. Therefore, if you have OLE capability, you most likely have this extension as well. As with any extension, you must first use it before you can make use of any of its functions and methods:

```
use Win32::OLE::Variant;
```

Creating a variant is done with the `Variant()` function:

```
$Var = Variant( $Type, $Value );
```

The first parameter (`$Type`) is the type of variant you are creating. This can be any one of the constants defined in [Table 5.1](#). This parameter might also be logically OR'ed with either or both of the values in [Table 5.2](#).

**Table 5.1. The Variant Data Types**

Data Type	Description
<code>VT_EMPTY</code>	This data type has no meaning other than void of any value. This is not the same as Perl's <code>undef</code> (to COM, <code>undef</code> is the equivalent of <code>VT_ERROR</code> ).
<code>VT_NULL</code>	There is no Perl equivalent for this variant type.
<code>VT_I2</code>	A signed short (2 bytes) integer.

**Table 5.1. The Variant Data Types**

<b>Data Type</b>	<b>Description</b>
VT_I4	A signed long (4 bytes) integer. This is the equivalent of C/C++'s <code>long</code> data type.
VT_R4	A float (4 bytes).
VT_R8	A double (8 bytes). This is the equivalent of C/C++'s <code>double</code> data type.
VT_CY	A 64-bit currency data type. This is not the same as a 64-bit integer. To get an integer value from this, you need to multiple its value by 10,000.
VT_DATE	A date data type (internally stored as a double).
VT_BSTR	A length-prefixed, null-terminated Unicode string. This is the only type of string that COM objects understand.
VT_DISPATCH	An <code>IDispatch</code> interface. Consider this to be the same as a <code>Win32::OLE</code> object.
VT_ERROR	An OLE <code>hrRESULT</code> value. This is what OLE and COM use to indicate results (like success or error values). This is the equivalent of Perl's <code>undef</code> .
VT_BOOL	A simple Boolean value.
VT_VARIANT	A reference to another variant.
VT_UNKNOWN	An <code>IUnknown</code> interface. This is similar to <code>VT_DISPATCH</code> except that there is no Perl equivalent. This is used internally.
VT_DECIMAL	A decimal-based value. This data type is not in the official list of allowable OLE Automation types of data, but most products support it.
VT_UI1	An unsigned character (1 byte)—not Unicode.

**Table 5.2. Variant Data Type Flags**

<b>Flag</b>	<b>Description</b>
VT_ARRAY	This flag indicates that the variant represents an array of values.
VT_BYREF	This flag indicates that the variant data type is a pointer to the actual data. In other words, the value held in the variant is a reference pointer to the data.

The second parameter of the `Variant()` function (`$value`) is the value that the result of the `Variant()` method will represent.

If the function succeeds, it returns a `Win32::OLE::Variant` object; otherwise, it fails and returns `undef`.

This function is scarcely ever used because almost all variant conversions are handled automatically. But for those who either have a specific need or want to play around with it, [Example 5.24](#) demonstrates its use.

### **Example 5.24 Using a variant**

```
01. use Win32::OLE;
02. use Win32::OLE::Variant;
03. my $Workbook = Win32::OLE->GetObject( "c:\\temp\\mytest.xls" )
04.           || die "Could not open the workbook.\n";
05. $Workbook->{Application}->{Visible} = 1;
06. $Workbook->{Application}->Windows( $Workbook->{Name} )-
>{Visible} = 1;
07. my $Sheet = $Workbook->Sheets( 1 );
08. my $Cells = $Sheet->Range("A1");
09. my $Var = Variant( VT_BSTR, "Oh what a happy day" );
10. $Cells->{Value} = $Var;
11. sleep(5);
12. $Workbook->Close();
```

You might have been thinking about this whole variant thing and wondering how arrays are handled. Arrays are interesting because, of course, they cannot be defined as a discrete parameter—they are a list of parameters. Arrays are converted into the OLE data type known as a **SAFEARRAY**. This is simply an array of variants. All of the elements within an array are first converted into variants and then the resulting list of variants are converted into a **SAFEARRAY**. Finally, a new variant is created of the data type **VT\_VARIANT | VT\_BYREF | VT\_ARRAY**.

You must note that multidimensional Perl arrays are converted into an array of equal elements. In other words, if you have an array consisting of two anonymous arrays (one with two elements and the other with three elements), the array will be converted into a 2 by 3 array. That is to say, it will consist of two arrays each with three elements. Any elements that had to be added to satisfy this transformation will consist of the **VT\_EMPTY** data type.

For example, the line

```
$ArrayRef = [
  [ 'array1_element1', 'array1_element2' ],
  [ 'array2_element1', 'array2_element2', 'array2_element3' ]
];
```

will be converted into the equivalent of

```
$ArrayRef = [
  [ 'array1_element1', 'array1_element2', undef ],
  [ 'array2_element1', 'array2_element2', 'array2_element3' ]
];
```

Arrays are interesting because, when you create one, you must define its dimensions. For example, you can define a simple variant string array:

```
$vArray = Variant( VT_ARRAY | VT_BSTR, 2 );
```

This will create a one-dimensional array of only two elements, although no data is stored in the array. The array is only created with enough space for two elements.

You can make a multidimensioned array just by passing in one parameter (indicating the number of elements to allocate) per dimension. For example a three-dimensional array can be created as:

```
$vArray = Variant( VT_ARRAY | VT_BSTR, 1, 1, 1 );
```

This results in a three-dimensional variant array in which each dimension of the array only has one element.

This array handling is what makes it possible to submit multidimensional arrays. [Example 5.25](#) demonstrates how arrays are handled for both passing into an object and retrieving from an object.

### **Example 5.25 Using arrays**

```
01. use Win32::OLE;
02. my $Workbook = Win32::OLE->GetObject( "c:\\temp\\mytest.xls" )
03.                 || die "Could not open the workbook.\n";
04. $Workbook->{Application}->{Visible} = 1;
05. $Workbook->{Application}->Windows( $Workbook->{Name} )-
>{Visible} = 1;
06. my $Sheet = $Workbook->Sheets( 1 );
07. $Sheet->Range("A1:C2")->{Value} = [
08.     [ qw( A1 B1 C1 ) ],
09.     [ qw( A2 B2 C2 ) ]
10. ];
11. my $ArrayRef = $Sheet->Range("A1:C2")->{Value};
12. foreach my $RowRef ( @{$ArrayRef} )
13. {
14.     print join(", ", @{$RowRef}), "\n";
15. }
or
01. use Win32::OLE;
02. my $Workbook = Win32::OLE->GetObject( "c:\\temp\\mytest.xls" )
03.                 || die "Could not open the workbook.\n";
04. $Workbook->{Application}->{Visible} = 1;
05. $Workbook->{Application}->Windows( $Workbook->{Name} )-
>{Visible} = 1;
06. my $Sheet = $Workbook->Sheets( 1 );
07. $Sheet->Range("A1:C2")->{Value} = [
08.     [ qw(( A1 B1 C1 ) ) ],
09.     [ qw(( A2 B2 C2 ) ) ]
10. ];
11. my( $Row1, $Row2 ) = @{$Sheet->Range("A1:C2")->{Value}};
12. print join(", ", @{$Row1}), "\n";
13. print join(", ", @{$Row2}), "\n";
```

### **Storing Data Using Put()**

After you have a variant, you might want to set the variant object with a value. To do this, you use the `Put()` method:

```
$VariantObject->Put( [$DimA [, $DimB [...]],] $$Value );
```

The first group of parameters is optional and only applies to variant arrays. Each of these parameters indicates the dimension numbers for the array element you are going to set with a value. For example, let's say you have a two-dimensional array, and each array consists of five elements. If you wanted to set the value for element 2,4 in Perl, you would do something like:

```
$PerlArray[2][4] = "Foo";
```

But with a variant you would use:

```
$VariantArray->Put( 2, 4, "Foo" );
```

The last parameter is the actual value you want to set. An attempt will be made to convert the value to an appropriate type. For example, if the variant is of the `VT_BSTR` type, then the numeric value 3.14 is converted to the string "3.14".

If the variant is an array, then you can pass in an array reference as a single parameter. This will let you set multiple array element variables at one time. Each element in the array reference can be either a value or another array reference. The latter enables you to set multidimensional arrays all at once.

To set a single dimension array, you could use:

```
$VariantArray->Put( [ "Hello", "Goodbye", "Thank you" ] );
```

For a multidimensional array, you could use:

```
$ArrayRef1 = [ 1,, 2, 3, 4, 5 ];
$ArrayRef2 = [ 6,, 7, 8, 9, 10 ];
$VariantArray->Put( [ $$ArrayRef1, $ArrayRef2 ] );
```

If the array is of type `VT_UI1`, then you can easily set the data of the entire array by passing in a Perl string into the `Put()` method. Make sure the array is allocated with enough array elements to hold the entire string:

```
$Data = "This is my data";
$VariantArray = Variant( VT_ARRAY | VT_UI1, length( $Data ) );
$VariantArray->Put( "This is my data" );
```

## ***Retrieving Data using Get()***

Just as you use the `Put()` method to store data into a variant, you can retrieve data using the `Get()` method:

```
$Value = $VariantObject->Get( [$DimA [, $DimB [, ...]]] );
```

The only group of parameters only applies to variant arrays. Each of these parameters indicates the dimension numbers for the array element value you are retrieving. For example, let's say you have a two-dimensional array, and each array consists of five elements. If you wanted to get the value for element 2,4 in Perl, you would do something like:

```
$Value = $PerlArray[2][4];
```

But with a variant you would use:

```
$Value = $VariantArray->Get( 2, 4 );
```

If the array is of type `VT_UI1`, then you can retrieve the data of the entire array as a Perl string by calling `Get()` without any parameters:

```
$Data = "This is my data";
$VariantArray = Variant( VT_ARRAY | VT_UI1, length( $Data ) );
$VariantArray->Put( "This is my data" );
$value = $VariantArray->Get();
```

## Type Libraries

The purpose of OLE is to enable an application to interact with another application even though they might be written by different programmers. This enables Outlook to interact with Photoshop and a fax printer driver to interact with your list of fax numbers held in Eudora. None of these programs was written with any other in mind, so none knows how to interact with any of the others. This is what OLE was designed to fix.

To make these programs work, they must be able to learn about each other's properties' and methods' prototypes. That is, if one program is going to use the `Open()` method on a COM object, it will need to learn what types of parameters the method takes. OLE applications create libraries of technical information that any other program can access. These libraries define the types of properties and methods (in addition to tons of other information) that exist in a particular class. The repositories of information are known as type libraries.

Typically, type libraries are those funny `.TLB` files that you find all over your hard disk. These are binary files that contain the information needed by an application to know how to handle a particular COM object.

Type library information does not have to be in a `.TLB` file, however. It can be embedded in a `.DLL`, `.OCX`, or even an `.EXE` file. They are also not required, so some OLE applications just do not have any type libraries.

The Microsoft COM/OLE Viewer tool can display various type libraries. It graphically shows the different interfaces that a particular library exposes. Additionally, it shows the constants, enum values, and structures that are used by the classes in the library. See the section called "[OLE and COM Object Documentation](#)" for more information.

## **Constants**

Just when you think you understand OLE and how to use it, a new twist reveals itself. When you program using the `Win32::OLE` extension, you will most likely run into the problem of needing to use constants. Look at line 186 of [Example 5.27](#) in the case study at the end of the chapter, where it sets a range of cells' horizontal alignment to `xlCenter`. That constant equates to some value that even the Excel documentation does not describe. (It only tells you to use the constant.) Unless you somehow discover the value, you will need to reference the `xlCenter` constant by name. This is where the problem comes in: Perl does not define the `xlCenter` constant; however, type libraries do.

Constants and their values are among all the information held within a type library. You can load the constants from a type library by using the `Win32::OLE::Const` module:

```
use Win32::OLE::Const [ $$TypeLibrary [, $VerMajor [, $VerMinor [,  
$Language ] ] ]  
];
```

The first (optional) parameter (`$TypeLibrary`) is a string to a type library. This is the name of a type library, as in "[Microsoft Word 8.0 Object Library](#)". You can specify regular expression wildcards to make it easier to match a library. This parameter is passed in to a regular expression as `/^$name/`, so you could specify a type library name of "`.*Basic`" to load the constants for "[Visual Basic for Applications](#)". If multiple versions of the same type library are registered, the most recent version (the latest version) will be used.

The second parameter (`$VerMajor`) is optional and represents the major version number. This would be a 5 for the Windows Media Player 5.2. By specifying this parameter, the constants from the type library will be loaded only if a library is found whose major version number matches this parameter.

The third parameter (`$VerMinor`) is optional and is used only if a value is passed in for the second parameter. (If `undef` is passed in as `$VerMajor`, this parameter is ignored.) This value represents the lowest minor number that is acceptable. By specifying a 5 for the second parameter and a 2 for this parameter, for example, a type library will be loaded only if it is version 5 and has at least a minor number of 2 (such as 5.2 or 5.3). Type libraries with a version of 5.1 or 6.0 will not be loaded.

The optional fourth parameter (`$Language`) represents a language ID. If this is specified, a library is loaded only if it has a matching language ID. You would want to use this if, for example, you needed to load the constants for the Turkish language. To make use of this parameter, you will need to use the `Win32::OLE::NLS` module. A description of this is beyond the scope of this chapter. If you need to use this, you should study the documentation embedded in the `NLS.PM` module.

If successful, the module will load, and the user's namespace will be populated with the constants from the specified type library. If unsuccessful, the module will load, but no constants will be loaded. [Example 5.25](#) demonstrates how the module is used in this way.

If you need to access constants but will not know the name of the type library until runtime, you could use the `Load()` method:

```
$Hash = Win32::OLE::Const->Load( ( $TypeLibrary | $Object ) [ ,  
$VerMajor [ ,  
$VerMinor [ , $Language ] ] ] );
```

The parameters are identical to those specified with the `use` command to load the module, with one exception. The first parameter can be either a string describing a type library or a Perl `Win32::OLE` object.

The `Load()` method will return a reference to a hash containing the type libraries' constants and values if successful. If the method fails, it returns `undef`. [Example 5.26](#) illustrates how this method is used.

### **Example 5.26 Loading constants at runtime**

```
01. use Win32::OLE;  
02. use Win32::OLE::Const;  
03. my $Name = shift @ARGV || "c:\\temp\\mytest.xls";  
04. if( my $Object = Win32::OLE->GetObject( $Name ) )  
05. {  
06.     my $Const = Win32::OLE::Const->Load( $Object );  
07.     my $iCount = 0;  
08.     print "Constants for ";  
09.     print Win32::OLE->QueryObjectType( $Object );  
10.    print ":\\n";  
11.    foreach my $Constant ( sort( keys( %$Const ) ) )  
12.    {  
13.        $iCount++;  
14.        print "$iCount) $Constant = '$Const->{$Constant}'\\n";  
15.    }  
16. }
```

### **Warnings**

There is a variable that can be set to handle warnings and errors relating to a COM object and Perl's interaction with it. The `$Win32::OLE::Warn` variable can be set to one of four values listed in [Table 5.3](#). Setting this variable will affect the way Perl contends with OLE errors.

**Table 5.3. Values for the `$Win32::OLE::Warn` variable**

<b>Value</b>	<b>Description</b>
0	Errors are ignored. Any interaction with a COM object that causes an error will return <code>undef</code> . Use the <code>Win32::OLE-&gt;LastError()</code> method to determine the error.
1	A warning message is displayed (the <code>Carp::carp()</code> function is called) only if the <code>\$^W</code> variable is set (as is done with the <code>-w</code> switch). This is the default value.

**Table 5.3. Values for the \$Win32::OLE::Warn variable**

Value	Description
2	A warning message is displayed (by means of the <code>Carp::carp()</code> function). Unlike the 1 value, the warning message is displayed regardless of the setting of <code>\$^W</code> .
3	An error message is displayed and the script terminates (by calling the <code>Carp::croak()</code> function).

## OLE and COM Object Documentation

One of the greatest dilemmas that faces OLE programmers is the nuisance of documentation. To know how a script can interact with a COM object, documentation on the object model of the OLE class is necessary. This can be difficult to find because most users do not need this information.

The best places to start looking for OLE information (in particular, the object model of a given class) are the help files. Microsoft has done a good job of publishing the object model for its major applications such as Office 97 in the help files. To find this documentation, you need to go to the help file and look up the Visual Basic for Applications section.

For all practical purposes, `Win32::OLE` can make use of OLE-related Visual Basic documentation (help files, books, magazines, and so forth). Such information describes the different objects, methods, properties, and constants.

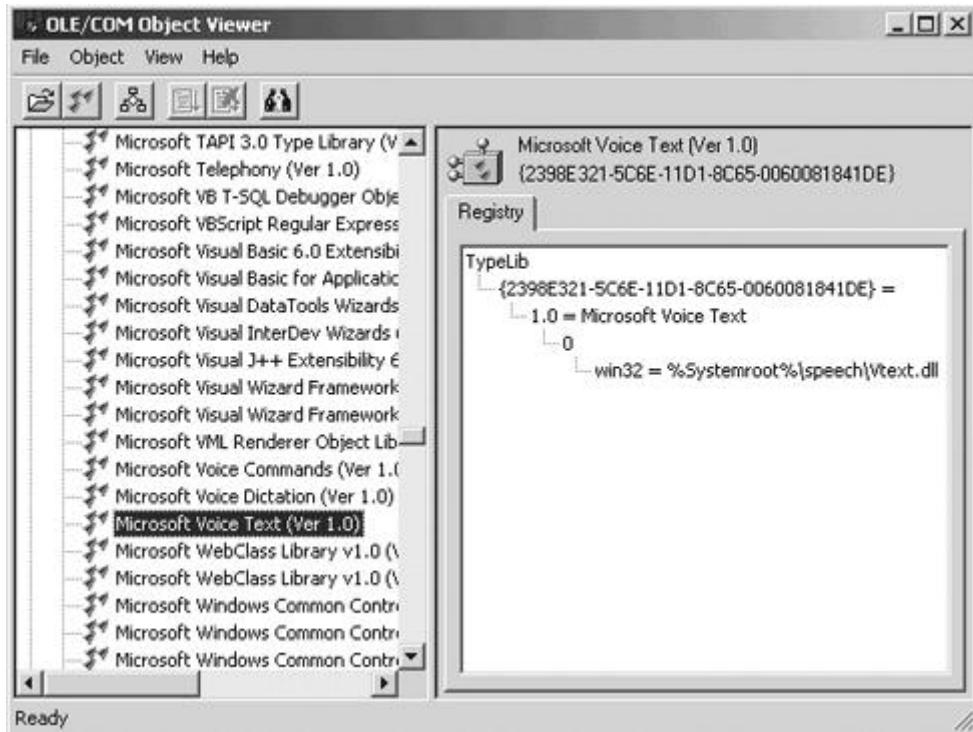
If you are looking for sources of documentation, you can find much in Software Development Kits (SDKs). For example Microsoft's NetShow, ADSI, and IIS SDK's come with help files that not only describe the object models but also give code examples. The code samples are fantastic to study and port to Perl.

There is one place to go for documentation that is usually overlooked: type libraries. Most all COM applications come with type libraries either embedded within their executable `.exe`, `.dll`, or `.cox` files or as a separate `.tbl` file. The problem is that this data is binary and difficult to read, so you would need a type library browser. It just so happens that Microsoft has one called OLE View (MSVC 6.0) or OLE/COM Viewer (MSVC 5.0). It comes with the developer environments (such as Visual C++, Visual Basic, or InterDev). You can also get a copy of it by downloading the Microsoft Platform SDK Tools, but this is a large download (around 12.3MB).

A type library browser will describe the different classes, methods, properties, enumerations, constants, properties, and other information. These descriptions are very descriptive indeed and include method parameter types and return values. As an added bonus, most type libraries have help strings that explain what a method, property, or constant is and how it is used. And to think that all of this information is right on your hard drive; it's just not easy for you to get to.

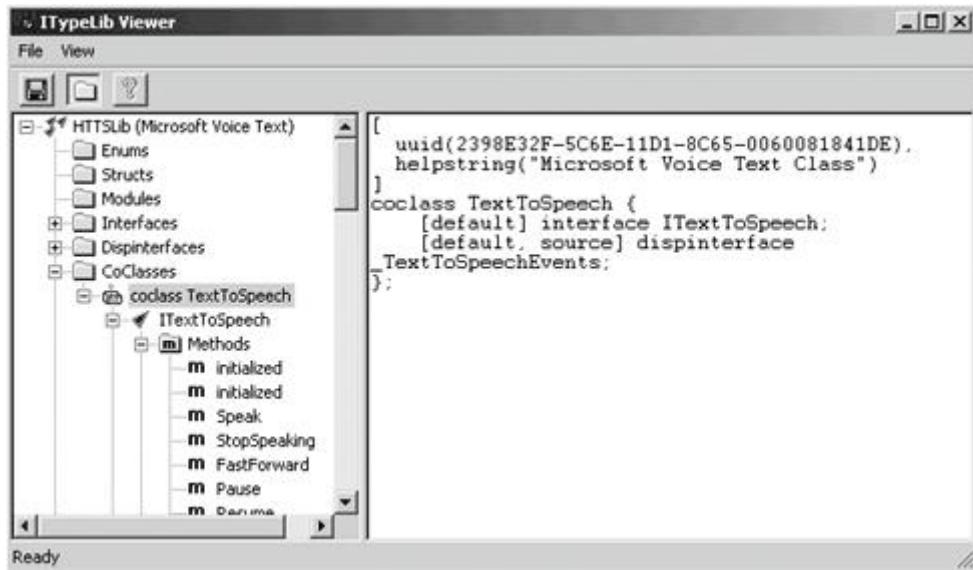
Consider [Figure 5.1](#). This is an example of using the OLE/COM Object Viewer. In this case, we have selected the type library called Microsoft Voice Text. If you double-click on this entry, another window appears. This window displays constant values (also known as *enums*) and various other information. What we are interested in are the COM classes, viewed as *CoClasses*.

**Figure 5.1. Using the OLE/COM Viewer.**



As [Figure 5.2](#) illustrates, the Microsoft Voice Text library contains a CoClass called `TextToSpeech`, and it consists of an interface called `ITextToSpeech`. Let's look at this interface. It has several methods, but there are three that I want to focus on.

**Figure 5.2. Examining dispinterfaces with OLE/COM Viewer.**



Carefully look at [Figure 5.2](#) and notice that the `ITextToSpeech` interface has two methods called `initialized`. Earlier in this chapter, I explained that, when using COM, a property is really just a method. Well, here is the proof. If you examined both of these methods, you would see one that states:

```
[id(0x00000001), propget, helpstring("property initialized")]
long initialized();
```

And another one that states:

```
[id(0x00000001), propput, helpstring("property initialized")]
void initialized([in] long rhs);
```

The first one is listed as a "propget," and the second is listed as a "propput." These mean, respectively, a get property (querying the value) and a put property (writing the value). The first one (the *get* property) is really just a method called `initialized()` that returns a `long` value. The second one is just a method called `initialized()` that accepts a `long` value and returns nothing. This is how properties work in COM; they are really just two methods. `Win32::OLE` and COM do all the work deciding what method to call. Your Perl script just handles these methods as if they were properties. Interestingly enough, if `initialized()` was simply a read-only property, it would only expose the "propget" version of the method. Because there would be no "propput" version, you would have a read-only property value.

You can also see that the `Speak()` method is defined as:

```
[id(0x00000002), helpstring("method Speak")]
void Speak(BSTR Text);
```

Because there is not a "propput" or "propget" setting, this method is considered a real method, not a property. Its prototype indicates that it accepts a text string (a `BSTR`, refer to [Table 5.1](#)). Really, this is the only method that I am interested in here because we want to have Perl speak out loud.

Examining [Figure 5.2](#) again, you'll notice that the `TextToSpeech` COM class has a UUID (class ID). This value is:

2398E32F-5C6E-11D1-8C65-0060081841DE

You can pass this in as a GUID to the `new()` function as in:

```
$Speech = new Win32::OLE( "{2398E32F-5C6E-11D1-8C65-
0060081841DE}" );
```

If this is successful, you can pass in any text string into the `$Speech->Speak()` method.

[Example 5.27](#) demonstrates how easy it is to go from discovering a type library (using OLE/COM Viewer) to writing an application. The speech engine used in this example comes standard with Windows 2000 and Windows ME. However, you can just as easily download it from Microsoft's Web site at <http://www.microsoft.com/speech/>.

### **Example 5.27 Playing with the *ITextToSpeech* interface**

```
01. use Win32::OLE;
```

```

02. my $Speech = new Win32::OLE( "{2398E32F-5C6E-11D1-8C65-
0060081841DE}" )
03.             || die;
04. my $NextTime = time();
05. while( 1 )
06. {
07.     my $Time = time();
08.     if( $Time >= $NextTime )
09.     {
10.         my @Date = localtime( $Time );
11.         # Handle some funky 24 to 12 hour conversions
12.         $Date[2] -= 12 if( 12 < $Date[2] );
13.         $Date[2] = 12 if( 0 == $Date[2] );
14.         # Make the voice sound okay for minutes < 10
15.         if( 0 == $Date[1] )
16.         {
17.             $Date[1] = "o'clock";
18.         }
19.         elsif( 10 > $Date[1] )
20.         {
21.             $Date[1] = "o' $Date[1]";
22.         }
23.         my $TimeString = sprintf( "The time is %s %s", $Date[2],
$Date[1] );
24.         print $TimeString, "\n";
25.         $Speech->Speak( $TimeString );
26.         # Don't bother the voice again for a minute
27.         $NextTime = $Time + 60;
28.     }
29.     # Take a snooze so we don't waste cpu time
30.     sleep( 1 );
31. }

```

## Case Study: Daily Administrative Network Reports

For many administrators who have a large server farm, it becomes difficult tracking each and every NT server. Ideally, the administrator would come in the morning and run the Event Viewer program. She would then connect to each server in her domain and look through the logs for any errors. But this is ideal, and in the real world, this is simply an unacceptable use of time.

A Perl script could be scheduled to run every morning (let's say an hour before the administrator comes in) that would scan the Net for all servers and then connect to each one's event log. It could then query for all errors that have occurred over the past 24 hours and dump them to an Excel spreadsheet.

Now all the administrator needs to do is come in every morning and check the administrator's workbook for the daily list of errors. This is a much quicker solution.

[Example 5.28](#) accomplishes this by using `Win32::OLE` to interact with Excel. It must load the workbook or create it if it does not already exist. The script creates a new worksheet and labels it with the current date. If one already exists with the same date, then the new sheet is given a name

with the date followed by a number such as "98.10.31 #2". The new worksheet is then populated with only errors and warnings from the event logs on different servers.

**Example 5.28 Using Win32::OLE to populate a spreadsheet with server errors**

```
01. use Win32::OLE qw( with );
02. use Win32::OLE::Variant;
03. use Win32::EventLog;
04. use Win32::NetAdmin;
05. use Win32::OLE::Const "Microsoft Excel";
06.
07. $Win32::EventLog::GetMessageText = 1;
08. $DateFormat = "DDD mmm dd, yyyy " hh:mm:ss";
09. $Class = "Excel.Application";
10. $Domain = "";
11.
12. @EVENT_SOURCES = ( "System", "Application" );
13. $EVENT_TYPE = EVENTLOG_ERROR_TYPE | EVENTLOG_WARNING_TYPE;
14.
15. $Dir = "c:\\temp";
16. $FileName = "AdminReport.xls";
17. $File = "$Dir\\$FileName";
18.
19. $SecPerDay = 24 * 60 * 60;
20. $Now = time();
21. $TimeLimit = $Now - ( $SecPerDay * 1 );
22.
23. $Excel = GetApplication( $Class ) || Error( "Could not start $Class" );
24. $Book = GetWorkbook( $Excel ) || Error( "Could not obtain a workbook" );
25. $Sheet = GetWorksheet( $Book ) || Error( "Could not obtain a worksheet" );
26.
27. print "Fetching list of servers...\n";
28. Win32::NetAdmin::GetServers( '', $Domain, SV_TYPE_DOMAIN_CTRL
| SV_TYPE_DOMAIN_BAKCTRL , \@Servers );
29.
30. $Row = 3;
31. foreach $Machine ( sort( @Servers ) )
32. {
33.     print "Processing $Machine.\n";
34.     $Sheet->Range( "A" . ++$Row )->{Value} = $Machine;
35.     $Sheet->Range( "A$Row" )->{Font}->{Bold} = 1;
36.     map{ ProcessLog( $Machine, $_, $EVENT_TYPE ); }
( @EVENT_SOURCES );
37.     $Row++;
38. }
39. ShutDownSheet( $Sheet );
40.
41.
42. sub ProcessLog
```

```

43. {
44.     my ( $Machine, $Source, $Type )= @_;
45.     my $Flag, $Num, %Hash;
46.     my $Event = new Win32::EventLog( $Source, $Machine ) || 
47.         sub
48.     {
49.         $Sheet->Range( "B" . ++$Row )->{Value} = "Unable to
connect.";
50.         return;
51.     };
52.     if( $Event->GetNumber( $Num ) )
53.     {
54.         my $iCount = 0;
55.         $Flag = EVENTLOG_BACKWARDS_READ | EVENTLOG_SEQUENTIAL_READ;
56.         do
57.         {
58.             if( $Event->Read( $Flag, $Num, \%Hash ) )
59.             {
60.                 print "\r $Source records processed: ", ++$iCount, " "
x 10;
61.                 if( $Hash{EventType} & $Type )
62.                 {
63.                     my ( $EventType, $Color, $Time );
64.
65.                     if( $Hash{EventType} == EVENTLOG_ERROR_TYPE )
66.                     {
67.                         $EventType = "Error";
68.                         $Color = 3; # Red
69.                     }
70.                     elsif( $Hash{EventType} == EVENTLOG_WARNING_TYPE )
71.                     {
72.                         $EventType = "Warning";
73.                         $Color = 53; # Red-Orange
74.                     }
75.                     elsif( $Hash{EventType} ==
EVENTLOG_INFORMATION_TYPE )
76.                     {
77.                         $EventType = "Information";
78.                         $Color = 1; # Black
79.                     }
80.                     $Row++;
81.                     # Format the time so that we can create a date based
variant
82.                     $Time = "" . localtime( $Hash{TimeGenerated} );
83.                     $Time =~ s/^.*?\s+(.*?)\s+(.*?)\s+(.*?)\s+(.*)/$1 $2
$4 $3/;
84.                     $Sheet->Range( "B$Row:H$Row" )->{Value} = [
85.                         "$Source: $EventType",
86.                         $Hash{Source},
87.                         ($Hash{Event}) ? $Hash{Event} : "None",
88.                         ($Hash{User}) ? $Hash{User} : "N/A",
89.                         $Hash{Computer},

```

```

90.         new Win32::OLE::Variant( VT_DATE, $Time ),
91.         $Hash{Message}
92.     ];
93.     $Sheet->Range( "B$Row" )->{Font}->{ColorIndex} =
94.         $Color;
95.     }
96.     # This will cause the next reading of the registry to
move to the
97.     # next record automatically.
98.     $Num = 0;
99. } while( $TimeLimit < $Hash{TimeGenerated} );
100. print "\n";
101. Win32::EventLog::CloseEventLog( $Event->{handle} );
102. }
103. }
104.
105. sub GetApplication
106. {
107.     my( $Class ) = @_;
108.     my( $Application );
109.
110.    if( ! ( $Application = Win32::OLE-
>GetActiveObject( $Class ) ) )
111.    {
112.        $Application = new Win32::OLE( $Class , "Quit" ) || die;
113.    }
114.    $Application->{Visible} = 1;
115.    return( $Application );
116. }
117.
118. sub GetWorkbook
119. {
120.     my( $Application ) = @_;
121.     my $Book;
122.     if( ! ( $Book = $Application->Workbooks( $FileName ) ) )
123.     {
124.         if( ! ( $Book = $Application->Workbooks->Open( $File ) ) )
125.         {
126.             my $Temp = $Application->{SheetsInNewWorkbook};
127.             $Application->{SheetsInNewWorkbook} = 1;
128.
129.             $Book = $Application->Workbooks->Add();
130.             $Book->SaveAs( $File );
131.
132.             $Application->{SheetsInNewWorkbook} = $Temp;
133.             $UseSheetNumber = 1;
134.         }
135.     }
136.     # Make the workbook window visible (Office 2000)
137.     $Book->{Application}->Windows( $Book->{Name} )->{Visible} =
1;

```

```

138.     return( $Book );
139. }
140.
141. sub GetWorksheet
142. {
143.     my( $Book ) = @_;
144.     my( $Sheet );
145.     if( $UseSheetNumber )
146.     {
147.         $Sheet = $Book->Worksheets( $UseSheetNumber );
148.     }
149.     else
150.     {
151.         $Sheet = $Book->Worksheets()->Add();
152.     }
153.     SetupWorksheet( $Sheet );
154.     return( $Sheet );
155. }
156.
157. sub SetupWorksheet
158. {
159.     my( $Sheet ) = @_;
160.     my( $Range );
161.     my( $Date, $Name, $iCount, @Date );
162.     @Date = localtime();
163.     $Date = sprintf( "%04d.%02d.%02d", $Date[5] ++ 1900,
$Date[4] ++ 1, $Date[3] );
164.     $Name = $Date;
165.     $iCount = 1;
166.     while( $Sheet->{Parent}->Worksheets( $Name ) )
167.     {
168.         $Name = "$Date #" . ++$iCount;
169.     }
170.
171.     $Sheet->{Name} = $Name;
172.     $Range = $Sheet->Range( "A1" );
173.     $Range->{Value} = "Server Error Logs for the morning of
$date";
174.     $Range->{Font}->{Size}=24;
175.     $Range->{Font}->{ColorIndex} = 6; # Yellow
176.     $Sheet->Rows("1:1")->{Interior}->{ColorIndex} = 5; # Blue
177.     $Range = $Sheet->Range("A3:H3");
178.     $Range->{Value} = [
179.         "Server", "Type", "Source", "Event", "User",
180.         "Computer", "Time", "Description" ];
181.     with( $Range->{Font},
182.           Bold => 1,
183.           Italic => 1,
184.           Size => 16
185.     );
186.     $Range->{HorizontalAlignment} => xlCenter;
187.     with( $Sheet->Columns( "G" ),

```

```

188.     NumberFormat => $DateFormat,
189.     HorizontalAlignment => xlCenter
190.   );
191. }
192.
193. sub ShutDownSheet
194. {
195.   my( $Sheet ) = @_;
196.   $Sheet->Columns( "B:G" )->AutoFit();
197.   $Sheet->Columns( "H:H" )->{ColumnWidth} = 72;
198.   $Sheet->Rows() ->AutoFit();
199.   $Sheet->{Parent}->Save();
200.   $Sheet->{Parent}->Close();
201. }
202.
203. sub Error
204. {
205.   my( $Error ) = @_;
206.   print "$Error\n";
207.   exit();
208. }

```

Notice that line 8 decides the date format that will be used in the worksheets. This works well for English systems, but it might need to change based on your locale. For example, people in Germany might want to change this to "ttt mmm tt, jjjj -- hh:mm:ss".

Line 1 loads the `Win32::OLE` requesting that the `with` method be exposed in the main namespace. Line 2 loads support for OLE variants, and line 5 will load the `Win32::OLE::Const` module exporting all of the constants related to Microsoft Excel.

Line 7 enables the `Win32::EventLog` extension to collect message text when the script reads an event log entry.

Lines 23 through 25 Call the functions `GetApplication()`, `GetWorkbook()`, and `GetWorksheet()`. These functions are defined by the script itself.

The `GetApplication()` function attempts to obtain a COM object of the specified class with a call to `Win32::OLE->GetActiveObject()` on line 110. If the object was not obtained, then a new COM object is requested with a call to the `new()` method on line 112. Line 114 sets the application's main window to be visible so the user can see what is happening.

The `GetWorkbook()` function attempts to load the spreadsheet for the administration report. This occurs on line 122 with a call to the application's `Workbooks($FileName)` method. Notice that the name of the file is passed in, not the full path to the file. This is because this line attempts to find the workbook in the list of currently open workbooks. The `Workbooks()` method will search the collection of open workbooks for one with a name that is passed in.

If the workbook is not currently open, then line 122 will fail and line 124 is given a chance to run. This line attempts to open the workbook specified by `$File`. If this fails, then line 127 sets the application's `SheetsInNewWorkbook` property to 1. This tells Excel that any new workbook to be

created will only have one spreadsheet. Then line 129 creates the new workbook with a call to the `Add()` method, and line 130 saves this new workbook using the path `$File`. If a workbook is created using `Add()`, then a global variable `$UseSheetNumber` is set to `1` for use in the call to `GetWorksheet()`.

The `GetWorksheet()` function will either request the particular worksheet specified by the global variable `$UseSheetNumber` (line 147) or create a new worksheet (line 151). Either way, the `SetupWorksheet()` function is then called that formats the worksheet.

Lines 162 through 169 figure out how to name the worksheet. Each worksheet is given a name based on the current date. The name follows the universal date format (like `1998.10.31`). This is created in line 163. However, if the program is run more than once a day, it will already have a worksheet with the current date as its name.

To avoid name collisions, lines 166 through 169 will query the workbook to see if a worksheet exists with the given name. If one does exist, then `$iCount` is increased, and the name is changed to reflect the date and the new count, such as `1998.10.31 #2`.

The script will loop between lines 166 and 169 until the call into `$Sheet->{Parent}->Worksheets( $Name )` does not return anything. If the worksheet exists, then the method call will return a COM object representing the specified name. This indicates that the worksheet exists.

Line 171 changes the name of the current worksheet. Then lines 172 through 190 format the worksheet, setting the appropriate font attributes, cell alignments, number formats, and colors.

The actual engine of the program is between lines 28 and 38. Line 28 retrieves a list of all currently online primary and backup domain controllers of the domain to which the computer running script belongs. For each machine, the `ProcessLog()` function is called. This function connects to the machine's event log (line 46), reads all event log entries in backward sequential order (starting with the most recent entry), and processes them until the first entry is retrieved that is not more recent than `$TimeLimit` – which is defined as one day before the current time (line 21 defines it as the current time minus 24 hours). Lines 84–92 use an anonymous array to store values into multiple cells of the spreadsheet.

## Summary

Although OLE applications have become quite important, building the COM-based infrastructure for Win32 Perl has been empowered with the capability to tap into this pool of resources—OLE, COM, or whatever you want to call the technology quickly becoming *the* interface for managing a Windows machine (in particular NT boxes). Technologies such as Active Directory Services Interface (ADSI) and Windows Management Instrumentation (WMI) both rely on COM (refer to Chapter 9 of *Win32 Perl Scripting: The Administrator's Handbook*).

Perl's `Win32::OLE` extension and its related modules not only give access to OLE-enabled applications, they also make COM interaction very easy by automatically handling variant creation and disassembly as well as dispatch marshalling. It is quite literally as easy to use COM objects in Perl as it is in Visual Basic. Actually, to some degree, it is easier.

The biggest problem with the `Win32::OLE` extension is that it is difficult to find documentation for different object models. Without such documentation, a COM object is like a black box; you can

see it, but you do not know what it does or how to use it. For the most part, this dilemma is resolved by vigilant research looking for SDK help files and help files from applications. Additionally, books and articles written about Visual Basic can give lots of insight into how to interact with COM objects.

`Win32::OLE` might well be the most important and impressive extension in the Win32 Perl arsenal. With patience and study, this extension can empower your code to interact with almost any OLE application, including almost any Win32-specific function.

## Chapter 6. Communication

To me, one of the coolest things a programmer can do is use a computer to communicate. Talking is what computers do best, and they do it constantly. Whether it is between the video card and the microprocessor, a program and another program, or one computer to another, this need to transfer information is commonplace.

Perl gives a script access to various communication interfaces such as sockets, anonymous pipes, file handles, and directory handles, to name a few. Native Perl does not, however, make use of Win32 communication needs such as sending network messages and using Win32 named pipes. This is where Win32 extensions, once again, come to the rescue.

This chapter covers the following extensions:

- `Win32::Message`
- `Win32::Pipe`

These extensions provide the capability to send messages across the network and to transfer data over named pipes from process to process and machine to machine.

### Sending Messages

The Win32 networking environment is based mostly on Microsoft's LAN Manager. This environment is based on the NetBEUI protocol, which is an extended version of the NetBIOS protocol. This protocol is fairly useful and quite efficient under typical LAN-based networks.

One of the features of NetBEUI is that each computer on the network has a unique name that differentiates the machine from other machines. This means that, for one computer to communicate with another computer, it would have to know the name of the machine. When a computer connects to the network, it makes an announcement that it exists and that it wants to use a particular name. If any computer already uses that name, it is up to the computer with the name to inform the new computer that the name is already taken. At that point, the computer will refuse to connect to the network. This process is known as *name registration*.

A computer can register two different types of names: a *unique name* that can be registered by only one computer and a *group name* that several computers can register.

When one computer wants to talk to another, it sends out a request onto the network, looking for the target computer. The request includes the network card's unique MAC address of the computer wanting to talk. This request is broadcast to every computer on the network. If the target computer hears this request, it will respond directly to the calling computer by using the caller's MAC address

found in the request. This response informs the calling computer of the destination computer's MAC address. After this name resolution has occurred, both computers know each other's MAC addresses and can talk directly to each other without having to send out broadcasts to every other computer on the network.

### Note

*A media access code (MAC) address is a unique, 12-digit, hexadecimal value that no other network card in the world shares. Network vendors work together to guarantee that this code is indeed unique for each network card produced. All network cards have a hard-coded MAC address, so it is not necessary to configure one. For some cards (such as IBM's Token Ring cards), however, an administrator can specify a different MAC address. This is sometimes known as a locally administrated address.*

When you send a message to a group of computers, the message is sent as a broadcast. When a computer hears the broadcast, if that computer has registered a group name that matches the name in the broadcast, the computer will process the message. All other computers in the network will ignore the message.

### Note

*Win32 platforms make use of other protocols such as IPX/SPX and TCP/IP. These protocols use other forms of techniques for broadcasting messages such as UDP datagrams, IP broadcasts, and multicasting (to name a few). Windows will use these various methods to emulate NetBEUI message passing. This enables a program or script to not worry about what protocol is used. It can be assured that the messaging will work correctly, regardless of protocol.*

When a computer registers a group name on the network, it is, in essence, proclaiming that it belongs to a particular group of computers. Windows for Workgroups took advantage of group names and used them to create workgroups. Later, when Windows NT was released, group names were used to represent domains.

Similar to unique computer and group names are usernames. A username is registered on a machine and represents the user who has logged on. As you might expect, a machine running services that require a user account to log on (such as a Web server or fax server) makes an attempt to register the username on the network. As with all unique names, however, if one exists already on the network, the attempt fails and no additional attempts are made. This is why, if you log on to Workstation A and then log on to Workstation B without first logging off of Workstation A, network messages sent to your username will go to Workstation A.

To be more accurate, a username is really a *messaging name*. This is a unique name that identifies a messaging service. When a user logs on, a *messaging name* based on the user's user ID is registered. Likewise, when a computer boots up, it registers a messaging name based on its computer name. Messaging names contain a special character that all network services recognize. You'll learn more about these special characters later.

The `Win32::Message` extension gives Perl the capability to manage these names and send messages. To receive sent messages on Windows 95, Windows 98, and Windows for Workgroups, the `WINPOPUP.EXE` program must be running. Windows NT requires that the messaging network service be installed and started; the NetBIOS interface must be installed as well.

### Note

*The `Win32::Message` extension does not work on Windows 95 machines. This is because the messaging API that this extension uses does not exist on Windows 95. Until Windows 95 supports the API, this extension is only supported from NT machines. However, any Windows machine (2000/NT, Windows 95/98/ME, and Windows for Workgroups) can receive messages sent by means of this extension.*

It is important to note the difference between a computer's messaging name and a user's messaging name. When a user logs on to a Windows NT, Windows 95, Windows 98, or Windows for Workgroups machine, the user's user ID is registered as a username. This allows someone to send a message to that user, but it does not allow someone to access shared resources using the username. If the administrator logs on to an NT machine that happens to be sharing its CD-ROM as `\machine\cdrom`, for example, you could not access that share as `\administrator\cdrom` because there is a difference between a username and a computer name. You could, however, send a message to `administrator` or `machine`.

### Note

*Like all extensions and modules, you should install the latest version of `Win32::Message`. Details on this can be found in [Appendix A](#), "Win32 Perl Resources."*

Microsoft allows a network name to be up to 15 characters long. When the name is registered on the network, however, it will always be 16 characters long. This is because the 16th character is special and indicates the type of name being represented. If you specify a network name that is less than 15 characters, the name will be padded with spaces. If you register a username of `JOEL`, the actual name registered on the network is as follows:

`JOEL_____<0x03>`

Notice that the four-letter name `JOEL` is padded with space characters (here they are shown as `"_"`) so that it is 15 characters long. Then the value `0x03` is placed in the 16th position. This value identifies this name as a messaging name. [Table 6.1](#) and [Table 6.2](#) illustrate the list of possible values with their meanings.

### Tip

*To see what various names are registered on your machine, you can run the `NBTSTAT.EXE` command using the `-n` or `-s` flags: `nbtstat -nnbtstat -S`.*

Typically, when the operating system loads, a unique network name is registered identifying the machine. Depending on which services are running on the machine, however, other network names are registered as well. They will use the same name, but the 16th byte will change reflecting the service. There might be a name registered indicating the computer's name, another indicating that the machine is a domain master browser, another indicating that NetDDE is running, and yet another showing that the machine is a RAS client. All in all, there could be five or more network names registered—all with the same name, differing only in the value of the 16th byte. See [Table 6.1](#) and [Table 6.2](#) for more information regarding the values of the 16th byte.

**Table 6.1. Unique NetBIOS Name Values**

16th Byte Value	Description
0x00	<b>Workstation service name.</b> This is also known as a computer name.
0x03	<b>Messenger service name.</b> A name with this character is used by the messenger service to send network messages. This value is appended to usernames and computer names.
0x06	<b>RAS server service.</b> Servers running the RAS service will have a network name with this value.
0x1B	<b>Domain master browser name.</b> A name with this value is a primary domain controller.
0x1F	<b>NetDDE service.</b> If a machine has NetDDE running, a computer name will be registered with this value.
0x20	<b>Server service.</b> A machine that runs the Server service (which allows other machines to access its files) will register a network name with this value.
0x21	<b>RAS client.</b> Any RAS clients will register a network name with this value.
0xBE	<b>Network monitor agent.</b> Any machine running Microsoft's network monitoring agent software (collecting network traffic statistics) will register a network name with this value.
0xBF	<b>Network monitor.</b> Machines running Microsoft's network monitoring (network sniffer) software will register a name with this value. This software comes with Windows 2000\XP Server and Systems Management Services (SMS).

**Table 6.2. Group NetBIOS Name Values**

16th Byte Value	Description
0x1C	A domain group.
0x1D	A master browser group name.
0x1E	A normal group name.
0x20	An administration group.
_MSBROWSE_	This is a special case. No value is placed specifically in the 16th byte; instead,

**Table 6.2. Group NetBIOS Name Values**

16th Byte Value	Description
_MSBROWSE_	is appended to a domain name. This is broadcast on a subnet announcing the domain to any master browser that might be listening.

### Tip

*All the functions in the Win32::Message extension can be performed on remote machines. To do this, however, the script must be running from an account with administrative privileges.*

## Registering Messaging Names on the Network

To register a unique messaging name on the network, you can use the `Win32::Message::NameAdd()` function:

```
Win32::Message::NameAdd( $Machine, $Name );
```

The first parameter (`$Machine`) is the name of the machine to which you are adding the specified name. The computer name can be prefixed with double backslashes or double forward slashes, as in `\Server` or `//Server`. Using an empty string rather than a computer name results in registering the name of the computer running the script.

### Note

*A NetBIOS network name can consist of any character, with the exception of the asterisk (\*).*

The second parameter (`$Name`) is the name you are adding. The name can consist of any (even nonprinting) characters with the exception of a `NULL` (a character with a value of 0), which would terminate the string. The string can be up to 15 characters long. Even though NetBIOS/NetBEUI names are 16 characters in length, Microsoft reserves the last character to identify the type of name (a user, machine, domain, workgroup, or some other type of name—refer to [Table 6.1](#) and [Table 6.2](#) for a list of the various 16th-character values).

If the name is successfully added, the function will return a `true` value; otherwise, it returns `false`. The function will fail if the name has already been added.

In [Example 6.1](#), a new network name is added to the local computer. Notice that the new name not only has a space and a single quotation mark, it also has a value of `0xFF` as the eighth character. There is no significance to the character `0xFF` in this case other than to illustrate that it can be done.

Notice that line 3 adds the new name by specifying an empty string as the computer name. This causes the local machine to register the specified name. If this string was a machine name, then an attempt to register the specified name on the local or remote machine is made. After the name is added, used, and no longer needed, it should be deleted. Deleting the name will be discussed a bit later.

## Note

The `Win32::Message::NameAdd()` function only registers message names. This means that if you register the name "MYNAME", it will be registered on the network as "MYNAME \x03". Notice that the 16th character is the value 0x03, indicating that the name is used for receiving network messages.

If you want to register a computer name as a machine's alias, refer to either `Win32::Message::NBNameAdd()` or `Win32::AdminMisc::ComputerAliasAdd()`.

### **Example 6.1 Adding a messaging name to a computer**

```
01. use Win32::Message;
02. my $Name = "Joel's \xFF Home";
03. if( Win32::Message::NameAdd( "", $Name ) )
04. {
05.     print "Successfully added the name '$Name'.\n";
06. }
```

In the case of [Example 6.1](#), a messaging name will be registered onto the network. The `NameAdd()` function that the example relies on only registers message names. However, if you run the `nbtstat.exe` command and display the NetBIOS name table, you will notice that there are additional registered names. These additional names (those with byte 16 values other than 0x03) cannot be registered using the `NameAdd()` function. This is due to a limitation of the Win32 API.

Interestingly enough, the Win32 API exposes low-level NetBIOS access. Using this functionality, the `Win32::Message` extension allows a script to register *any* network name. This is done using the `NBNameAdd()` and `NBNameGroupAdd()` functions:

```
Win32::Message::NBNameAdd( $Name );
Win32::Message::NBNameGroupAdd( $Name );
```

These two functions both attempt to register the specified name on the network. The `NBNameAdd()` function will register a unique network name. However, the `NBNameGroupAdd()` function will attempt to register a group network name. Multiple machines can be registered to this same group network name.

The only parameter (`$Name`) is the NetBIOS name you want to add. Unlike the `NameAdd()` function, this parameter must be a full NetBIOS name. This means that the name specified *must* be 16 characters long. The parameter can consist of any character values including embedded null characters (0x00).

If the parameter is more than 16 characters, then only the first 16 characters are used. If fewer than 16 characters are specified, then the string is padded with spaces.

If the unique NetBIOS name is successfully registered on the network, then the function returns a `TRUE` (1) value. Otherwise, it returns `FALSE`.

Names added using the `NBNameAdd()` and `NBNameGroupAdd()` functions will exist only throughout the duration of the process. When the Perl script terminates, these names are removed from the machine.

### **Warning: NBxxxx() Functions Accept Any NetBIOS Name**

*A word of warning about using the various `NBxxxx()` functions. They all accept any NetBIOS name. You can specify a name with any character including lowercase and terminating nulls. This can be a problem because Win32 network functions all assume that network names are uppercase. If you add a name such as "My Fun House \x03", don't expect to get any network messages from it because network messages sent out using the Win32 API will always capitalize the network name.*

Example 6.2 illustrates how this function is used. It is almost identical to Example 6.1, except that in line 2 the NetBIOS name is a full 16 characters long. Notice that the 16th character is set to the value of `\x21` `\x21`, which indicates a RAS client. Even if the machine does not have RAS installed, it will look as if it does.

### **Example 6.2 Adding a NetBIOS name to a computer**

```
01. use Win32::Message;
02. my $Name = "Joel's \x00 Home \x21";
03. if( Win32::Message::NBNameAdd( $Name ) )
04. {
05.   print "Successfully added the name '$Name'.\n";
06. }
```

## **Listing Registered Names**

Because you can add messaging names to your machine, it is only a matter of practicality that you should be able to discover what messaging names your machine has registered. You can discover this information with `Win32::Message::NameEnum()`:

```
Win32::Message::NameEnum( $Machine );
```

The only parameter (`$Machine`) is the name of the machine from which you want to generate the list. This computer name can be prepended with either forward slashes or backslashes. If this parameter is an empty string, the computer running the script is used.

The `NameEnum()` function will return an array of computer names and usernames that the machine has registered on the network. There is no other way to discern which is a username and which is a computer name except guessing.

[Example 6.3](#) shows how you can use `NameEnum()` to get a list of computer messaging names from both your local computer as well as a remote machine.

### **Example 6.3 Discovering registered messaging names**

```
01. use Win32::Message;
02. push( my @MachineList, ( Win32::NodeName(), @ARGV ) );
03. foreach my $Machine ( @MachineList )
04. {
05.     my @List = Win32::Message::NameEnum( $Machine );
06.     print "Registered message names for: $Machine\n";
07.     map { print "\t$_\n" } @List;
08. }
```

Just as there is a way to add any NetBIOS name, there is a function that enables a script to list all the registered NetBIOS names. This is similar to the `NameEnum()` function; however, it will provide a list of *all* names, not just messaging names. The function is called `NBNameEnum()`:

```
Win32::Message::NBNameEnum( $Machine, \@List );
```

The first parameter (`$Machine`) is the name of the machine from which you want to generate the list. This must be a full, 16-character computer name. If the name specified is less than 16 characters long, it is assumed to be a computer name and will be converted to uppercase, padded with spaces, and a `\x00` value will be set for the 16th character. *This parameter does not accept a double backslash prepended to the name unless the name indeed has two backslashes in it.*

The second parameter is a reference to an array. If the function is successful, this array is populated with hashes consisting of the keys listed in [Table 6.3](#). Each hash's `Flags` key contains two separate values: type and status.

To determine the NetBIOS name's *type*, logically AND the Flag key's value with the constant `NB_NAME_FLAG_TYPE_MASK` and refer to [Table 6.4](#). To determine the name's *status*, logically AND the Flag key's value with the constant `NB_NAME_FLAG_STATUS_MASK` and refer to [Table 6.5](#).

**Table 6.3. NBNameEnum's Array Hash Keys**

Key	Description
Name	The 16-character NetBIOS name.
Flags	Flags indicating the type and state of the name. See <a href="#">Table 6.4</a> and <a href="#">Table 6.5</a> .
Number	The NetBIOS name's entry in the NetBIOS name table.

**Table 6.4. A NetBIOS Name's Types**

Flag Value	Description
UNIQUE_NAME	Indicates that the NetBIOS name is a unique name. Only one such name can exist on

**Table 6.4. A NetBIOS Name's Types**

Flag Value	Description
	the network.
GROUP_NAME	Indicates that the NetBIOS name is a group name. Multiple machines can register for the same name.

**Table 6.5. A NetBIOS Name's States**

State	Description
REGISTERING	The name is currently being registered on the network.
REGISTERED	The name has successfully been registered on the network.
Deregistered	The name has been marked for removal. After all active sessions to the name are closed, the name will be removed.
Duplicate	During registration of the name, a duplicate was found already on the network.
Duplicate_Dereg	During registration of the name, a duplicate was found on the network. However, the existing name has been marked for removal.

**Example 6.4** uses the `NBNameEnum()` function to get an exhaustive list of NetBIOS names that are registered on a particular machine. The script accepts any number of machine names (don't prepend double backslashes to the computer names). The script effectively emulates the `nbtstat.exe -n` command.

Notice that line 13 calls `NBNameEnum()` by converting the specified computer name to uppercase. This is important because the various `NBNamexxxx()` functions are case sensitive. Win32 registers computer names in all uppercase (unless, of course, you use `NBNameAdd()` to register the name).

Line 20 checks to see if the network name is a unique or group name. Here we use a shortcut by logically ANDing the Flag key's value with `GROUP_NAME`. If the result equates to `GROUP_NAME`, then the flag indicates a group name; otherwise, it is a unique name. The proper way to do this is to AND the Flag key's value with `NB_NAME_FLAG_TYPE_MASK`. The resulting value is either `GROUP_NAME` or `UNIQUE_NAME`, indicating the type of network name.

Line 21 figures out the status of the network name. This is done by ANDing the Flag key's value with `NB_NAME_FLAG_STATUS_MASK`. The resulting value corresponds to a value listed in [Table 6.5](#).

#### **Example 6.4 Discovering a computer's registered network names**

```
01. use Win32::Message;
02. my @MachineList = (scalar @ARGV) ? @ARGV : (Win32::NodeName());
03. my %STATUS = (
04.     &REGISTERING => 'Registering',
05.     &REGISTERED => 'Registered',
06.     &Deregistered => 'Deregistered',
07.     &DUPLICATE => 'Duplicate',
```

```

08.    &DUPLICATE_DEREG => 'Duplicate Dere registered'
09. );
10. foreach $Machine ( @MachineList )
11. {
12.     my @List;
13.     next unless( Win32::Message::NBNameEnum( uc $Machine,
14.     \@List ) );
15.     $~ = "NameHeader";
16.     write;
17.     foreach $Name ( @List )
18.     {
19.         $Name->{Name} =~ s/^(\.{15})(.)/sprintf( "%s<%02x>" , $1,
unpack( 'C', $2 ))/e;
20.         $Name->{Type} = ($Name->{Flags} & GROUP_NAME) ? "GROUP"
21.         :"UNIQUE";
22.         $Name->{Status} = $STATUS{ $Name->{Flags} &
23.         NB_NAME_FLAG_STATUS_MASK } ;
24.         write;
25.     }
26.     format NameHeader =
27. Registered names for @<<<<<<<<<<<<<<<<<<<<<<
28. $Machine
29.     Name                Type          Status
30.     -----
31. .
32. .
33. format NameData =
34.     @<<<<<<<<<<< @<<<<<< @<<<<<<<<<<<<<<<<<
35.     $Name->{Name} , $Name->{Type} , $Name->{Status}
36. .

```

## Removing Names from a Machine

Along with adding and listing network names, the `Win32::Message` extension supports the capability to remove names you have added to a machine. The `Win32::Message::NameDel()` function removes a message name on either a remote machine or your local machine:

```
Win32::Message::NameDel( $Machine, $Name );
```

The first parameter (`$Machine`) is the name of the machine you from which want to remove a username. If this parameter is an empty string, the local machine will be used. The machine name can be prepended with double forward slashes or backslashes if desired (but these are not required).

The second parameter (`$Name`) is the name to be removed. This name is not case sensitive and can support any character except a `NULL` value, just like the `NameAdd()` function.

If the function is successful, the return value is `TRUE`, and the specified machine will deregister the name from the network. This will enable other computers to successfully register the name. The specified machine will no longer respond to messages sent to the name that has been removed.

The `NameDel()` function will remove *only* message names that you already have added from the specified computer. Any attempt to remove a nonmessaging name (such as computer names, group names, and so forth) will result in the function failing.

[Example 6.5](#) demonstrates adding a new messaging name to a machine. If the name is successfully registered on the network, it is removed (line 8). The last part of the code attempts to remove the registered computer name (line 21). This will fail; otherwise, you might have found a bug in the OS!

### **Example 6.5 Removing messaging names from a machine**

```
01. use Win32;
02. use Win32::Message;
03. my $Machine = Win32::NodeName;
04. my $NewName = "Joel's Home";
05. if( Win32::Message::NameAdd( $Machine, $NewName ) )
06. {
07.     print "Successfully added the name '$NewName'.\n";
08.     if( Win32::Message::NameDel( $Machine, $NewName ) )
09.     {
10.         print "Successfully removed the name '$NewName'.\n";
11.     }
12. else
13. {
14.     print "Unable to remove the name '$NewName'.\n";
15. }
16. }
17. else
18. {
19.     print "Unable to add the name '$NewName'.\n";
20. }
21. if( Win32::Message::NameDel( $Machine, $Machine ) )
22. {
23.     print "Successfully removed the computer name '$Machine'.\n";
24.     print "This should never, EVER happen!\n";
25. }
26. else
27. {
28.     print "Unable to remove the computer name '$Machine'.\n";
29.     print "This is expected and what should occur.\n";
30. }
```

Similar to `NameDel()` is the `NBNameDel()` function. This function will attempt to remove *any* NetBIOS name:

```
Win32::Message::NBNameDel( $Name );
```

The only parameter (`$Name`) is a 16-character NetBIOS name you want to remove. Unlike the `NameDel()` function, this parameter is case sensitive. Because you can specify any value for the 16th character, it is possible to deregister any name.

This function only works on the local machine, so you cannot remove network names remotely. Additionally, only a process that created a NetBIOS network name can remove it using this function.

[Example 6.6](#) demonstrates using the `NBNameDel()` function.

### ***Example 6.6 Removing any NetBIOS name from a machine***

```
01. use Win32::Message;
02. my $Name = "~~UniqueName~~ \xla";
03. if( Win32::Message::NBNameAdd( $Name ) )
04. {
05.     print "Successfully added the name '$Name'.\n";
06.     if( Win32::Message::NBNameDel( $Name ) )
07.     {
08.         print "Successfully removed the name '$Name'.\n";
09.     }
10. else
11. {
12.     print "Unable to remove the name '$Name'.\n";
13. }
14. }
```

## **Sending Messages Between Machines**

Usernames are very useful when you need to send messages to and from a computer. A message can be sent to a specific user, a computer, or a group. When a message is received on an NT machine, a window appears, displaying the message along with who sent it and when it was sent. A Windows 95/98/ME or Windows for Workgroups machine will display the message only if the `WINPOPUP.EXE` program is running. If you have ever printed to a network printer and received a message informing you that the print job was successfully printed, you have witnessed the wonder of sending messages.

To send a message, you can use the `Win32::Message::Send()` function:

```
Win32::Message::Send( $Machine, $Receiver, $Sender, $Message );
```

The first parameter (`$Machine`) is the name of a computer that will send the message. This can be any computer name, but you must have the required permissions to force the remote machine to send the message. If this parameter is an empty string, the local machine will send the message. This is handy if an administrator needs to send a message from a server, but he is on another machine.

The second parameter (`$Receiver`) is the name of the receiver. This can be a computer name or a username and can have prepended forward slashes or backslashes (but these are not required).

The third parameter (`$Sender`) is a bit of an oddity. It *should* be an empty string, but it does not have to be. If you supply anything other than the machine that sends the message (whatever is passed in as the first parameter), however, the function will fail, and the message will not be sent. This limitation is imposed by the operating system, supposedly to prevent someone from sending a message claiming to be from another source (this is a form of *spoofing*).

The fourth parameter (`$Message`) is the message to be sent. This can be a message (either text or binary) of any size. There is no imposed limit; however, a receiver might have a size limit. Windows for Workgroups and Windows 95/98/ME, for example, both use the `WINPOPUP.EXE` application to send and receive messages. This application might have a limit to how many characters can be displayed. Windows NT, however, does not seem to have a limit for receiving messages.

If the function is successful, the message is sent to the receiver, and the return value is `TRUE`. If the function fails, the return value is `FALSE`.

### Tip

*To prevent spoofing, Win32 does not allow a script to send messages claiming to be from another machine. However, Windows 2000/XP machines enable a script to specify an alternative sender to other Windows 2000/XP machines.*

If the `Send()` function returns `TRUE`, the message has been sent, but there is no guarantee that the message was successfully received by the recipient. There is no way to know for sure whether the message went through.

### Tip

*You can send a message to all users on a computer, domain, or workgroup by specifying the name and appending an asterisk (\*). For example, if you use `Win32::Message::Send( "", "ACCOUNTING*", "", "The file server is going down at 10:00");`, everyone in the ACCOUNTING domain (or workgroup) will receive the message. Likewise, if you specify a computer name appended with an asterisk, the message will be sent to every user on that machine. This is handy if you are using a multiuser version of Windows, such as Citrix's WinFrame or Terminal Services, that allows multiple users to be logged on at the same time.*

[Example 6.7](#) shows how you can send a message to all Windows NT Servers in a specified domain. This example makes use of the `Win32::NetAdmin::GetServers()` function, which is described in [Chapter 2](#), "Network Administration."

### **Example 6.7 Sending a message to all NT machines in a domain**

```
01. use Win32::Message;
02. use Win32::NetAdmin;
03. my $Sender = Win32::NodeName();
04. my @List;
05. my $Message = <<EOM;
06. Don't forget to log off of your machine tonight.
```

```

07.    We are updating everyone's profiles and you
08.    need to be logged off.
09. EOM
10. if( Win32::NetAdmin::GetServers( "", "", SV_TYPE_SERVER,
\n@List ) )
11. {
12.    foreach my $Machine ( @List )
13.    {
14.        Win32::Message::Send( '', $Machine, $Sender, $Message );
15.    }
16. }

```

## Named Pipes

Another form of interprocess communication (IPC) is the named pipe. A *pipe* is a way of communicating between processes. You can think of a pipe much like the tin can and string telephones that children play with. One child speaks into a tin can. The sound travels across the string to the other end where her friend is listening. Pipes are very similar in that one process writes into one end of the pipe and the other process reads from the other end.

This is a handy way for two processes or even different threads in the same process to pass data to each other. You see this type of communication every time you download a Web page using your browser. The server process on some remote machine writes Web page data to one end of a TCP/IP socket, and the browser reads the data at the other end of the socket. The socket is essentially a pipe.

There are really two types of pipes: anonymous and named. *Anonymous pipes* are usually used within an application. For example, one thread might read data from different files and write it into one end of the pipe. Another thread might read the data from the pipe and process it somehow (maybe printing it or saving it to a tape backup device). This type of pipe is known as an anonymous pipe because there is no way to identify the pipe from another pipe. The program will have to keep track of what pipe it is using for what reasons.

*Named pipes* are different from anonymous pipes in that there is a name associated with the pipe. Therefore, as long as another process or thread knows the name of the pipe, it can connect to it and read from or write to the pipe. An example of such a named pipe is Microsoft's SQL Server. It creates a named pipe and listens to it for database queries. An application that wants to look up data from the database would connect to this pipe, submit a query, and then listen to the pipe for results. SQL Server provides a name to which the querying application tries to connect.

Really, there is not much difference between named pipes and anonymous pipes other than a named pipe has a name associated with the client end, which allows other processes to connect to it. The Win32 port of Perl does not provide native support for named pipes, only anonymous pipes. This is why the `Win32::Pipe` extension was written.

### Note

*Win32 pipes are not the same as named pipes from other operating systems. A named pipe created on a Win32 machine might not necessarily allow for non-Win32 pipe access. Don't be surprised if a script you write that creates a Win32 named pipe does not interoperate with a client running on a UNIX machine.*

In the Win32 world, the named pipe's name is shared among server processes. This means that if you create two named pipes using the same name, what really happens is that the first script creates the named pipe and the second script *thinks* it created a pipe but really is using the pipe already created.

After a pipe is created, the server process creates an *instance* of the pipe using the `Connect()` method. An instance of a named pipe is a particular connection between a client process and a server process. Think of a named pipe as a help desk. There are several "help desk engineers" (server processes) waiting for phone calls from confused users. When a user (the client process) calls the help desk phone number, a connection will be made to only one engineer. Even though each frustrated user is calling the same phone number (the named pipe), the user (client) is connected to a particular engineer (server). This telephone connection is an instance of the phone call (similar to an instance of the named pipe). This means you can have multiple server processes all waiting for connections on the same named pipe. The Win32 OS will decide which call was waiting first for a connection and will arbitrate all connections.

There are no limits to the number of instances of a named pipe that can exist, and there is no limit to how many different named pipes can exist on a machine.

### Tip

*When creating a named pipe, you are not limited to any namespaces. This means you can create a named pipe that looks like the following instances:*

```
\ServerName\pipe\MyPipe  
\ServerName\pipe\Stats\Logons  
\ServerName\pipe\Stats\Logoffs
```

*Notice in the last two examples that the named pipe resides in what appears to be a directory called Stats. The rules for naming named pipes are the same as those that govern naming files.*

*The pipe names are just literally names of pipes. Even though the examples show a Stats directory, no such directory really exists on any hard drive. Additionally, no Stats directory ever had to be created.*

## The Client/Server Paradigm and Named Pipes

Just like an anonymous pipe, the named pipe works by using a client/server model. That is, a server creates the pipe and listens to one end. A client then attaches and either talks or listens on the other end of the pipe. From the client's point of view, the pipe can act just like a file handle. As a matter in fact, a client can access a named pipe by opening the pipe as if it were a file using the `open()` function, as shown in [Example 6.8](#).

### **Example 6.8 A process connecting to a named pipe as a client**

```
01. use Win32;  
02. my $User = Win32::LoginName();
```

```

03. my $Node = Win32::NodeName();
04. my $Time = localtime();
05. my $PipeName = "\\\server\pipe\Logs";
06. if( open( FILE, "> $PipeName" ) )
07. {
08.   print FILE "User $User on machine $Node logged on at
$Time.\n";
09.   close( FILE );
10. }
11. else
12. {
13.   print "Unable to open the pipe due to error $!.\\n";
14. }

```

The client process does not need to know anything specific about the named pipe or how the named pipe was created; all it needs to do is open a file and treat it as if it were indeed a file. [Example 6.8](#) accesses a named pipe called "`\server\pipe\Logs`". For the sake of this example, assume that some process has created the pipe and is waiting for some other process to write data to it. When the process hears incoming data, it will save the data to a log file. The example shows how a client makes a connection to this named pipe. Notice how the connection is made by using the `open()` function as if it were opening a regular file. As far as the script is concerned, the named pipe is exactly that, a regular file.

### **Tip**

*When you create a server- or client-side pipe using `Win32::Pipe`, it is automatically in binary mode. No character interpretation occurs during these pipe transmissions.*

*If you open a client side of a named pipe using the Perl `open()` function, however, you might want to use the `binmode()` function to make sure that all data transferred in and out of the pipe is in binary mode.*

## **Types of Named Pipe States**

In the Win32 world, two different types, or states, of named pipes exist: byte and message. If a named pipe is set to the *byte* state, data is sent through the pipe as a stream of bytes. Data is sent one byte at a time and is received as a stream of bytes. This is similar in nature to streams that you might expect when you open a file. The byte state is used to send large amounts of data or data that is unpredictable in size, such as when you are sending data files.

In the *message* state of a named pipe, all data is sent over the pipe in discrete-sized messages. If you want to send more data than fits into the message size, you must send multiple messages. Messages are useful when a client must send a packet of information that is formatted and in a predictable size—for example, if you need to send a data structure or a simple message. The size of a message is determined by the buffer size of the named pipe.

### **Note**

*When dealing with named pipes, the terms "mode" and "state" are thrown around quite interchangeably. This is not by design but by circumstance.*

*The Win32 API makes references to states as well as modes representing the same thing. In some cases, there might be a constant such as PIPE\_READMODE\_BYTE that represents the read state of a named pipe. The fact that the word MODE is part of the constant would lead one to believe that it represents a mode and not a state. That would, however, be an incorrect belief.*

*Keep in mind that when you see the terms "mode" or "state," you should consider them to be the same.*

After a pipe has been created, the state of the pipe cannot be changed. A process can, however, change the way that the data is read from the pipe. There is a difference between the state of a pipe and the read state of a process's connection to a pipe. The former has already been discussed, but the latter refers to how the process will read data from the pipe.

If a pipe is in the message state, a client process can set the read state of its end of the pipe to either the byte or the message state by using the `State()` method. Only pipes created in the message state allow for this. Pipes created in the byte state can be read only as byte pipes. You learn more about the `State()` method later.

A few methods allow for quick and efficient transactions between a client and server, but they require that the read state of the pipe be in the message state.

## Note

*All named pipes are of the blocking type. This means that when a named pipe is waiting for a process to connect to the other end of the pipe (the process is listening) or data is being either sent or received over the pipe, all Perl processing waits. The `Win32::Pipe` extension does not take advantage of nonblocking, async-named pipe capabilities.*

## Creating Named Pipes

A process creates a named pipe object with the `new()` function:

```
$Pipe = new Win32::Pipe( $Name[, Timeout[, $State[, $Perms ] ] ] );
```

The first parameter (`$Name`) is the name of the pipe. This name will be accessed using a full UNC in which the share name is "pipe" and the pipe's name is appended to the end of the UNC as if it were a filename. If a name of "LogFiles" were passed in as the pipe's name, for example, a client would connect to it as "\server\pipe\LogFiles". The name can consist of any characters that can make up a valid Win32 filename—any characters except for the following:

:<> "\ / | ? \*

The second parameter (`$Timeout`) is optional and represents the timeout in milliseconds. This value specifies how long the pipe will wait for a client to connect before giving up. [Table 6.6](#) describes constants you can use for this parameter. The default value (if none is specified) is the equivalent to the `DEFAULT_WAIT_TIME` constant.

The optional third parameter for the `new( )` function (`$State`) is both the read and write states of the named pipe. This value can be a combination of a single value from [Table 6.7](#) logically OR'ed with a single value from [Table 6.8](#). If nothing is specified, `PIPE_TYPE_BYTE` and `PIPE_READMODE_BYTE` are used.

The optional fourth parameter is a security identifier (SID) or `Win32::Perms` object. The permissions that this parameter represents are applied to the new pipe. Refer to [Chapter 11](#), "Security," for more details on security and permissions.

**Table 6.6. Named Pipe Timeout Values**

Value	Description
<code>NMP_USE_DEFAULT_WAIT</code>	The operating system's default timeout value.
<code>NMP_NOWAIT</code>	Do not wait at all. If the function cannot be processed immediately, just give up.
<code>NMP_WAIT_FOREVER</code>	This will wait forever.

**Table 6.7. Pipe Read States That Can Be Specified During a Named Pipe's Creation**

Pipe State	Description
<code>PIPE_READMODE_BYTE</code>	The named pipe will be created in byte read mode. This will read data from the pipe as a stream of bytes, just as one would read from a file. This is usually used when data is expected from the pipe (such as sending binary file data).
<code>PIPE_READMODE_MESSAGE</code>	The named pipe will be created in message read mode. This will read data from the pipe in discrete messages of a fixed size. A typical application of this would be if the process expects announcements or commands from another process.

**Table 6.8. Pipe Write States That Can Be Specified During a Named Pipe's Creation**

Pipe State	Description
<code>PIPE_TYPE_BYTE</code>	The named pipe will be created in byte mode, which means data is sent across the pipe as a stream of bytes. This stream can be of any size in length.
<code>PIPE_TYPE_MESSAGE</code>	The named pipe will be created in message mode, sending data across the pipe as messages of a fixed and predictable size.

If the pipe was successfully created, the return value is a `Win32::Pipe` object; otherwise, `undef` is returned.

In [Example 6.9](#) and [Example 6.10](#), a named pipe is created called "My Test Pipe". It has a timeout value of 10,000 milliseconds (10 seconds) and is a message type of pipe, so the process will read data from the pipe in the message mode. Notice that [Example 6.10](#) requires that the `Win32::Perms` extension be installed.

### ***Example 6.9 Creating a named pipe***

```
01. use Win32::Pipe;
02. my $pipe = new Win32::Pipe( "My Test Pipe",
03.                           10000,
04.                           PIPE_READMODE_MESSAGE |
05.                           PIPE_TYPE_MESSAGE
06.                           ) || die;
```

### ***Example 6.10 Creating a named pipe with security***

```
01. use Win32::Pipe;
02. use Win32::Perms;
03. my $Perms = new Win32::Perms() || die "Unable to create a
Perms object.";
04. $Perms->Allow( "Marketing\\JOEL", READ );
05. $Perms->Allow( "administrator", FULL );
06. my $Pipe = new Win32::Pipe( "My Test Pipe",
07.                           10000,
08.                           PIPE_READMODE_MESSAGE |
09.                           PIPE_TYPE_MESSAGE,
10.                           $Perms
11.                           ) || die;
```

## **Connecting to Named Pipes**

A client process can connect to a named pipe in two different ways: by creating a named pipe connection object or by just opening the named pipe using the `open()` function described in [Example 6.8](#).

By creating a client-end `Win32::Pipe` object, however, you can control many aspects of the pipe itself; these aspects are discussed in the "[Changing Named Pipe States](#)" section later in this chapter.

To connect to a named pipe using a `Win32::Pipe` object, you create a client end (that is, you connect to an already existing pipe) using the `new()` command:

```
$Pipe = new Win32::Pipe( $Name[, Timeout[, $State ] ] );
```

The first parameter (`$Name`) is the full UNC of the named pipe, such as `\server\pipe\LogFiles`. The UNC will always begin with the machine name and a share name of `pipe`.

The second parameter (`$Timeout`) is optional and represents the timeout in milliseconds. This value specifies how long the pipe will wait for a client to connect before giving up. [Table 6.6](#) described constants you can use for this parameter. The default timeout value (if one is not specified) is equivalent to the `DEFAULT_WAIT_TIME` constant.

The optional third parameter (`$State`) for the `new()` command is the read state of the pipe. Unlike the third parameter used when creating a named pipe, this parameter is only the read state of the pipe. (Because the pipe has already been created, the pipe's state has already been determined.) This can be any single value previously documented in [Table 6.7](#).

Even though the operation for opening the named pipe is carried out in the same way that a server process creates a named pipe, when the `Win32::Pipe` extension sees that a full UNC has been specified as the first parameter, the pipe object created will be a client end.

## Tip

*When a client process connects to a named pipe that is on the same computer, a period (.) can be used rather than a computer name in the UNC. The following two UNC's are the same, for example, if the client is located on the machine called FileServer:*

```
\FileServer\pipe\LogFiles  
\.\pipe\LogFiles
```

*When a script is accessing a named pipe located on the same machine, the use of the period in lieu of the machine name is much faster because the OS will not have to go through the network stack.*

*On the other hand, it might be detrimental because it renders the script useless if it is run from another machine.*

A client process can also connect to a named pipe using Perl's `open()` function, as in [Example 6.8](#). This is much easier, but it limits the functionality of the client to only reading and writing data. Using the `open()` function this way enables you to specify read-only, write-only, or read/write access on the named pipe, just as if a regular file were being opened.

## Listening for Clients

When a server process creates a named pipe, it must then wait for a client to connect. After the connection is made, processing can continue, and data can be sent and received over the pipe. The server process will listen for a connection by using the `Connect()` method:

```
$Pipe->Connect();
```

The script will wait on this method until either a client connects to the pipe or the timeout value specified when the pipe was created is exceeded.

The function returns `TRUE` if a client is connected to the other end of the pipe; otherwise, `FALSE` is returned.

## Note

A client would never need to call the `Connect()` method. Only the server process uses this method because a client opening the client end of a named pipe has already connected to it.

## Sending Data Across Named Pipes

After a named pipe instance has been initiated and the server and client processes are connected, data can flow across the pipe. The `Win32::Pipe` extension creates full-duplex pipes so that both the server and client can both send and receive data.

When a process wants to send data across the pipe, there are a few different ways of doing this by using either the `transact()` or `Write()` method. The most obvious choice is to use the `Write()` method:

```
$Pipe->Write( $Data )
```

The `$Data` parameter refers to the data to be sent.

The method `Write()` writes the data to the named pipe and returns a value of `true` if it successfully sent the data; otherwise, it returns `false`.

Both [Example 6.10](#) and [Example 6.11](#) illustrate the use of the `Write()` method.

## Note

*It is possible when a process uses the `Write()` method to send data, but the other process might be delayed in reading the data.*

*The method will return when the data was successfully sent to the other end of the pipe. That does not mean the other process has read the data, however. The data might have been stored in a buffer, waiting to be read.*

Another way of sending data is by using the `Transact()` method. This not only will send data to the other end of the pipe, it will wait for a reply and return whatever is received. There is no difference between this and calling `Write()` followed by the `Read()` method other than convenience and speed. `transact()` is a bit faster than the `Write()/Read()` combination (some sources benchmark it at 10 percent faster). The `transact()` method, however, can only be run on a pipe created in the message state and if the read mode is message:

```
$Pipe->Transact( $SendData, $ReadData );
```

The first parameter (`$SendData`) is the data you are sending to the other process.

The second parameter (`$ReadData`) will receive the data that is sent back to the process over the pipe.

The `transact()` method will wait up to the timeout value specified when the `Win32::Pipe` object was created. The return value is the data received from the pipe.

The `TRansact()` method can be used by either client or server process; however, because a server process typically receives data, processes it, and then sends the results back to the client, a server process usually has no need to use this method.

The method will send all the data you pass into the method over the named pipe. This means that the message size is the size of the data you submit. When reading, it will read all the data received in one call. This means the receive buffer must be able to hold all the received data.

In [Example 6.9](#), the `TRansact()` method was used to submit a message to the server process. The method then returned after receiving a message from the server process. Notice that when the client end of the pipe was connected, the read state of the pipe was set to `PIPE_READMODE_MESSAGE`, which was required for the `transact()` method to work.

Another, and more typical, way of receiving data from a named pipe is by means of the `Read()` method:

```
$Pipe->Read();
```

The `Read()` method will wait until data has been read before returning. The return value is the data that was read from the pipe.

An example of the `Read()` method is found in [Example 6.10](#), which is the equivalent to the code found in [Example 6.8](#) and [Example 6.9](#). Because the `Read()` method does not require any specific read mode, no optional parameters are passed in during the connection to the client end of the pipe.

The `Read()` method waits until it receives data from the pipe. If there is no data waiting in the pipe's buffer, the `Read()` method waits until either the data is received or the other end of the pipe is disconnected (or closed). This can cause a problem if you need to check for any data that might be sitting in the pipe. If there is no such data, your script will sit waiting, possibly for a long, long time.

To avoid this time lag, your script can peek ahead and see whether there is anything in the pipe by using the `Peek()` method:

```
$Pipe->Peek( $Size );
```

The `$Size` parameter is the maximum number of bytes to return.

If the `Peek()` method is successful, it returns up to `$Size` bytes of data; otherwise, `undef` will be returned if either there is nothing waiting in the pipe or an error occurred.

## Note

*It is important to note that `Peek()` will not remove any data from the pipe, so if `Peek()` returns any data, you might want to actually retrieve the data from the pipe using the `Read()` method. The data returned by `Peek()` is the same data that will be returned by a call to `Read()`—except that `Read()` will return all the data, whereas `Peek()` will only return up to the specified number of bytes.*

The `Peek()` method is very useful when you must monitor several pipes for processing. This method can be used to check each pipe for data without waiting on the pipe, as in [Example 6.11](#).

### **Example 6.11 Monitoring pipes for processing using the `Peek()` method**

```
01. use Win32::Pipe;
02. my $Pipe = new Win32::Pipe( "My Test Pipe",
03.                             NMP_WAIT_FOREVER,
04.                             PIPE_TYPE_MESSAGE
05.                             | PIPE_READMODE_MESSAGE ) || die;
06. my $Pipe2 = new Win32::Pipe( "My Test Pipe",
07.                             NMP_WAIT_FOREVER,
08.                             PIPE_TYPE_MESSAGE
09.                             | PIPE_READMODE_MESSAGE ) || die;
10. my $fFlag = 1;
11. my $fFlag2 = 1;
12. print "Waiting for first client to connect...\n";
13. $Pipe->Connect();
14. print "Waiting for second client to connect...\n";
15. $Pipe2->Connect();
16. print "Running...\n";
17. while( $fFlag && $fFlag2 )
18. {
19.     $fFlag = Process( $Pipe ) if( $Pipe->Peek( 100 ) );
20.     $fFlag2 = Process( $Pipe2 ) if( $Pipe2->Peek( 100 ) );
21. }
22. $Pipe2->Close();
23. $Pipe->Close();
24.
25. sub Process
26. {
27.     my( $Pipe ) = @_;
28.     my( $Data ) = $Pipe->Read();
29.
30.     if( $Data =~ /quit/i )
31.     {
32.         return 0;
33.     }
34.     print "Received: $Data\n";
35.     return 1;
36. }
```

### **Example 6.12 A client for accessing the server from Example 6.11**

```

01. open( PIPE, "+< \\\\.\\pipe\\My Test Pipe" ) || "die Could
not connect: $!\n";
02. # Turn on autoflush – the same as removing file buffering.
03. select( PIPE );
04. $| = 1;
05. select( STDOUT );
06. my $Message = "This is my test message.\n";
07. print PIPE $Message;
08. while( my $Input = <STDIN> )
09. {
10.     print PIPE $Input;
11. }
12. print PIPE "quit\n";
13. close( PIPE );

```

**Example 6.13 A Win32::Pipe-based client for accessing the server from Example 6.11 using the *Transact()* method**

```

01. use Win32::Pipe;
02. my $Input;
03. my $Pipe = new Win32::Pipe( "./pipe/My Test Pipe",
04.                             NMP_WAIT_FOREVER,
05.                             PIPE_READMODE_MESSAGE ) || die;
06. my $Message = "This is my test message.";
07. $Pipe->Transact( $Message, $Input );
08. while( my $Input = <STDIN> )
09. {
10.     print PIPE $Input;
11. }
12. print $Pipe->Write( "quit" );
13. $Pipe->Close();

```

**Example 6.14 Another Win32::Pipe-based client for accessing the server from Example 6.11**

```

01. use Win32::Pipe;
02. my $Pipe = new Win32::Pipe( "./pipe/My Test Pipe" ) || die;
03. my $Message = "This is my test message.";
04. $Pipe->Write( $Message );
05. while( my $Input = <STDIN> )
06. {
07.     print PIPE $Input;
08. }
09. print $Pipe->Write( "quit" );
10.
11. $Pipe->Close();

```

## Disconnecting a Named Pipe from a Client

The purpose of a server process creating a named pipe is to allow client processes to connect to it. If a client successfully connects and performs whatever transactions are needed, the server process

will consider the session with the client finished. At this point, the server will need to disconnect from the client.

Disconnecting means the server process kicks the client off the named pipe. After this is done, the server can neither read from nor write to the pipe because there is no client on the other end. The server will want to call the `Connect()` method, which will wait for another client to connect to the pipe.

Many users of named pipes think the server process should close the pipe. This indeed would effectively disconnect the client process from the pipe, but it would also literally close the pipe so that the server no longer has access to it. So long as the server process needs to use the pipe, it should not be closed; instead, it should be disconnected:

```
$Pipe->Disconnect( [$Purge] );
```

The only parameter is optional. If it is nonzero, all data will be flushed from the pipe before it is disconnected. The method call will wait until the flush is completed before returning. If, for any reason, the flush takes a long time (maybe the client process is taking its time reading the data), the script will wait until all data has been flushed before continuing.

If the function is successful, the client process is disconnected from the named pipe, and the method returns `true`; otherwise, `false` is returned, and there is no guarantee that the client is disconnected.

It is interesting to note that a client process can call `Disconnect()`, but it should not. If a client process calls it, the process's connection to the named pipe is closed, but the Perl named pipe object remains active.

[Example 6.15](#) demonstrates, among other things, the `Disconnect()` method. This example is a server process that creates a new named pipe and waits for clients to connect to it. A client will write some data to the pipe, which this example will read and print out. Notice that, in line 37, the server process calls the `Disconnect()` method, which forces the client process to close its connection to the pipe. At this point, the pipe has only the server process on one end, and the other end is not connected to anything. The script then loops back to line 11 and waits for another client to connect.

### ***Example 6.15 A server process that logs user submissions***

```
01. use Win32::Pipe;
02. $LogFile = "users.log";
03. open( LOG, "> $LogFile" ) || die "Could not open $LogFile:
$!.\n";
04. select( LOG );
05. $| = 1;
06. select( STDOUT );
07. my $Pipe = new Win32::Pipe( "My Test Pipe",
08.                             NMP_WAIT_FOREVER,
09.                             PIPE_TYPE_MESSAGE
10.                            | PIPE_READMODE_MESSAGE ) || die;
11. while( $Pipe->Connect() )
12. {
13.     my $In = $Pipe->Read();
```

```

14. if( ! $In )
15. {
16.     # We received no data...something is wrong so
17.     # give up and wait for another connection
18.     # This happens when the client terminates it's
19.     # connection without telling us.
20.     next;
21. }
22. $LastUser = $User;
23. $LastTime = $Time;
24. @Info = $Pipe->GetInfo();
25. $User = $Info[2];
26. $Time = time();
27. print LOG "$User:$Time:$In\n";
28. $Count++;
29. print "$Count) $User at " . localtime( $Time ) . "\n";
30. $Message = "You are user number $Count.";
31. if( $Count > 1 )
32. {
33.     $Message .= " The previous user was $$LastUser at "
34.                 . localtime( $LastTime );
35. }
36. $Pipe->Write( $Message . "\n" );
37. $Pipe->Disconnect();
38. }
39. END
40. {
41.     $Pipe->Close();
42.     close( LOG );
43. }

```

## Closing Named Pipes

After a pipe has been used and is no longer needed, it must be closed, just like a regular file. Every pipe needs to be closed when no longer needed, whether it is a server end or a client end of the named pipe. The `Close()` method performs this action:

```
$Pipe->Close();
```

After the pipe is closed, the pipe object is no longer valid, and any attempt to use its methods will fail.

## Changing Named Pipe States

If a named pipe was created as a message type pipe (not a byte type), both the server and client ends of the pipe can be set to the read mode to make the pipe appear as if it is a message or byte type of pipe. This is typically used before any calls to the `transact()` method are made because that method will only work on pipes whose read mode is of the message type. The method to change the read mode of a pipe is the `State()` method:

```
$Pipe->State( [$ReadMode] );
```

The optional `[$ReadMode]` parameter specifies which read mode is desired.

The `state()` method will return the current state that the read mode is in. If a parameter was passed in, the return value reflects the new state of the pipe. If no parameters are passed in, the method retrieves only the current state. If a parameter is passed in, an attempt to change that state is made.

The only way to know whether the method was successful is to check the return value. The value will be the same as the desired state.

## Retrieving Named Pipe Information

Information pertaining to the named pipe can be of value to a script. Through the `GetInfo()` method, you can obtain information that details the user connected to the other end of the pipe and how many instances of the named pipe exist:

```
$Pipe->GetInfo();
```

The `GetInfo()` method returns an array that consists of five elements. [Table 6.9](#) lists these elements.

**Table 6.9. Array Elements Returned From the `GetInfo()` Method**

Element	Description
Element 0	The state of the pipe. This value actually contains other data that is not of any use to Perl (such as whether the named pipe uses nonblocking, asynchronous IO that is not implemented in this extension). To determine the state (or mode) of the pipe, logically AND element 0 with <code>PIPE_TYPE_MESSAGE</code> . If the result of the logical AND is 0, the mode of the pipe is <code>PIPE_TYPE_BYTE</code> .
Element 1	The number of instances. This number reflects how many instances of the named pipe currently exist in the OS.

### Element Description

Element 2	The client user ID. This is the user ID of the process connected to the client end of the pipe.
Element 3	This indicates how many bytes of data the OS collects before actually sending the data over the pipe to the other end. The data will be sent either if the OS has collected this number of bytes or if any amount of data has been waiting for at least the number of milliseconds indicated in element 4.
Element 4	This indicates how much time the OS waits before sending any data across the pipe in milliseconds. The data will be sent either if the OS has waited this long or if there are at least the number of bytes waiting to be sent as indicated in element 3.

## Checking and Resizing a Named Pipe Buffer

When a named pipe is created, a buffer is allocated that can accept incoming data. If a server process creates a named pipe, it has a buffer (as does a client who just connects to an existing named pipe). You can check the size of the buffer with the `BufferSize()` method:

```
$Pipe->BufferSize();
```

The returned value is the size of the pipe's buffer.

If you want to change the buffer size, you can use the `ResizeBuffer()` method:

```
$Pipe->ResizeBuffer( $Size );
```

The only parameter (`$Size`) is the size, in bytes, of the new buffer. Any existing buffer, and its data, will be destroyed before the new buffer is allocated.

Both methods return the size of the current buffer, or the size of the new buffer in the case of `ResizeBuffer()`.

## Establishing Connectionless Access to a Named Pipe

The `Win32::Pipe` extension has support for non-connection-based client access to a named pipe. If all you need to do is send some data over an existing named pipe and receive the server's reply, there is no need for the overhead associated with actually opening a client connection to the pipe. Instead, you can use the `CallNamedPipe()` function:

```
Win32::Pipe::CallNamedPipe( $Name, $Data, $ReadBuffer );
```

The first parameter (`$Name`) is the name of the named pipe. This is a full UNC that can use either forward slashes or backslashes. If the server end of the named pipe is on the same computer as the client end, you can use a period (.) as the computer name, as in `\.\pipe\My Test`.

The second parameter (`$Data`) is the data to be sent to the server end of the pipe.

The third parameter (`$ReadBuffer`) is a buffer that will be filled with the data received from the server.

If the `CallNamedPipe()` function is successful, it returns `TRUE`; otherwise, it returns a `FALSE` value. The `CallNamedPipe()` function is a quick and easy way to perform a simple transaction across a named pipe. It requires that the named pipe be of a message type (created using the `PIPE_TYPE_MESSAGE` option), however. It will make a connection to the named pipe and, if successfully connected, send `$Data`. Then it will read from the pipe and put the retrieved data into `$ReadBuffer`. [Example 6.16](#) shows how `CallNamedPipe()` can be used to make a powerful but very small script.

### **Example 6.16 Using the `CallNamedPipe()` function**

```

01. use Win32::Pipe;
02. if( Win32::Pipe::CallNamedPipe( "//server/pipe/My Test Pipe",
03.                                "Test",
04.                                $Buffer ) )
05. {
06.   print "$Buffer\n";
07. }

```

## Setting and Retrieving Named Pipe Win32 File Handles

The `Win32::Pipe` extension abstracts Win32 named pipes from Perl so that they are a bit easier to handle without having to contend with the details of Win32 file handles. If you are willing to deal with Win32 file handles, however, a new world of possibilities opens up.

You can retrieve the named pipe's Win32 file handle from the pipe object using the `GetHandle()` method:

```
$Pipe->GetHandle();
```

The value returned is a Win32 file handle. This value can be used in other functions that accept Win32 file handles, such as `Win32::AdminMisc::CreateProcess()`. This particular function enables you to specify a Win32 file handle for the new process's `STDIN` or `STDOUT`.

Another related function will either set or retrieve a standard handle. In a Win32 console application, the `STDIN`, `STDOUT`, and `STDERROR` handles can be retrieved or set using the `StdHandle()` method:

```
$Pipe->StdHandle( $Type[ , $Handle ] );
```

The first parameter (`$Type`) is the type of standard handle. This is one of the following values:

- `STD_INPUT_HANDLE`. The standard input handle
- `STD_OUTPUT_HANDLE`. The standard output handle
- `STD_ERROR_HANDLE`. The standard error handle

The second optional parameter (`$Handle`) is the Win32 file handle that is to be set to the standard file handle specified in the first parameter.

The return value is the Win32 file handle for the specific standard handle. If you are setting the handle, the new value of the handle is returned.

### Tip

*The `Win32::AdminMisc::CreateProcessAsUser()` function enables you to create a process that maps the standard file handles to valid Win32 file handles. This is remarkably cool because you can open a named pipe and map it to `STDIN`, `STDOUT`, or `STDERROR`. All you need to do is pass the return value from `$Pipe->GetHandle()` into the function. There is a catch, however: If you specify one standard file handle, you must specify all three of the standard handles (a Microsoft-imposed requirement). This is where the `StdHandle()` method comes in.*

You can make calls to `StdHandle()` and get the Win32 file handle for each standard handle you want to pass into the `CreateProcessAsUser()` function.

## Summary

Messaging and named pipes are simple and easy ways to pass data from one machine to another. The named pipes, in particular, provide a wonderful way to transfer data in full duplex between not only processes but machines as well. Because both messages and named pipes work independently from network protocols, you are not limited to using them only on TCP/IP networks, as sockets are so limited.

A named pipe can transfer any text or binary data. This can be data obtained or created by a script, read from a file, or queried from a database. One of my original uses of the named pipe extension was to interface a series of Microsoft Access and FoxPro ODBC databases on one machine with a Web server on another machine.

# Chapter 7. Data Access

For most administrators, their jobs take their Perl programming into realms of maintaining user accounts and managing servers. More and more administrators are finding it important to write scripts that interact with databases, however, whether for web-based common gateway interface (CGI) scripts or for querying administrative databases for reports.

Many Perl modules and extensions support databases, such as the variations of `xDBM_File` and libraries for SQL Server, Sybase, Oracle, MySQL, Dbase, and various others. A couple of Win32-specific extensions, however, marry the Win32 platform with generic database services such as ODBC. This chapter provides an overview on using the `Win32::ODBC` extension.

Other data access Perl extensions are available that provide similar functionality to `Win32::ODBC`, but each of them could easily consume a chapter themselves. Because `Win32::ODBC` comes standard with ActivePerl, it is covered here. One of the more popular alternatives is the Perl Database Interface (DBI). Tim Bunce and Alligator Descartes have written an excellent book on this subject, *Programming the Perl DBI* (O'Reilly & Associates). If you are exploring DBI as a means to access databases (even ODBC-based data sources) you will want to look into this book.

## What Is ODBC?

All databases have their own unique way of doing things. When a programmer wants to access database services from within his code, he has to learn how to use that particular database. Really, this is not so difficult because most of the major database vendors document their systems and provide libraries to which a coder can link. This is all fine and good as long as you always use one particular database. The moment you need to access a different type of database, however, not only will you have to learn another entire set of commands and procedures, you will also need to change your scripts so they can interface with this new database. Usually this means a total rewrite of the code.

Now imagine you want to write a database application that would work with any database your client might have. A perfect example is that you want to write a shopping cart CGI script for the web. Because you do not know which database your client might have implemented, you have to

write several variations of the same script to handle all the possible databases. If this is not more than you want to contend with, just imagine what it would be like to support all those scripts. Testing them would be just as horrific because you would have to install each and every database system.

Wouldn't it be nice if all databases conformed to just one standard so that if you programmed your scripts to utilize it, one script would work across all database systems? This is where Open Database Connectivity (ODBC) comes in.

ODBC is an initiative that the database industry has come to accept as the standard interface. Many people believe that ODBC is a Microsoft product, but they are incorrect in believing so. Microsoft did champion the API and was one of the first companies to have working implementations of it. The actual interface standard was designed, however, by a consortium of organizations, such as X/Open SQL Access Group, ANSI, ISO, and several vendors, such as IBM, Novell, Oracle, Sybase, Digital, and Lotus, among others.

The standard was designed to be a platform-independent specification, and it has been implemented on the Win32, UNIX, Macintosh, and OS/2 platforms to name just a few. ODBC has become so widely accepted that some vendors—such as IBM, Informix, and Watco—have designed their database products' native programming interface based on ODBC.

## **ODBC Models**

When a program uses ODBC, it just makes calls into functions defined by the ODBC API. When a Perl script makes a call into the ODBC API, it is typically calling in to a piece of software known as the ODBC Manager. This manager is a dynamic link library (DLL) that decides how to handle the call. Sometimes the ODBC Manager will perform some task, such as listing all available Data Source Names (DSNs), and return to the calling script. Other times, it will have to load a specific ODBC driver and request that the driver (usually another DLL) perform the task, such as connecting to the database and executing a query. Each level of software that the ODBC Manager has to pass through to accomplish a task is known as a tier.

ODBC has different tiered models that describe how the basic infrastructure works. Each model is premised on how many tiers exist between the

ODBC Manager and the actual database. There could, in theory, be an unlimited number of tiers, but the impracticality of administrating and configuring so many tiers renders only three common models: the one-, two-, and three-tier models.

### ***One-Tier Model***

The one-tier model consists of only one step (or tier). The client application talks to the ODBC Manager, which asks the ODBC driver to perform the task. (This is the first tier.) The driver opens the database file.

This is a very simple model in which the ODBC driver performs the work of database lookup and manipulation itself. Examples of single-tier ODBC drivers include Microsoft's Access, FoxPro, and Excel ODBC drivers.

### ***Two-Tier Model***

Just like the one-tier model, the client application talks to the ODBC Manager, which talks to the ODBC driver. (This is the first tier.) Then the ODBC driver will talk to another process (usually on another machine via a network) that will perform the actual database lookup and manipulation. (This is the second tier.) Examples of this are IBM's DB2 and Microsoft's SQL Server.

### **Three-Tier Model**

Just like the first two models, in the three-tier model, the client application talks to the ODBC Manager, which talks to the ODBC driver (tier one). The ODBC driver then talks to another process (usually running on another machine) that acts as a gateway (tier two) and relays the request to the database process (tier three), which can be a database server or a mainframe such as Microsoft's SNA server.

These different tiers are not as important to the programmer as they are to the administrator who needs to configure machines with ODBC installed. However, understanding the basic infrastructure helps in making decisions such as how to retrieve data from the driver so that network traffic is held to a minimum, as discussed later in the section "[Fetching a Rowset](#)."

## **Data Source Names**

To truly separate the client application from the database, there are Data Source Names (DSNs). DSNs are sets of information that describe to the ODBC Manager what ODBC driver to use, how the driver should connect to the database, what user ID is needed to log on to the database, and so forth. This information is collectively referred to by using a simple name, a Data Source Name. I might set up a DSN that tells the ODBC Manager that

I will be using a Microsoft SQL Server on computer `\DBServer`, for example, and that it will be accessed via named pipes using a user ID of 'JoeUser'. All this information I will call "Data."

When writing an application, I will just make an ODBC connection to the DSN called "Data," and ODBC will take care of the rest of the work. My application needs only to know that I connect to "Data" and that's it. I can later move the database to an Oracle server, and all I need to do is change the ODBC driver (and, of course, configure it) that the "Data" DSN uses. The actual Perl script never needs to be altered. This is the beauty of ODBC.

ODBC is an application programming interface (API), not a language. This means that when you are using ODBC, you are using a set of functions that have been predefined by the groups that created the ODBC specification. The ODBC specification also defines what kinds of errors can occur and what constants exist. So ODBC is just a set of rules and functions.

Now, when you use ODBC, you need to interact somehow with the database. ODBC provides a set of rules and functions on how to *access* the database but not how to *manipulate data in* the database. To do this, you need to learn about a data querying language known as SQL.

## **SQL**

The language that ODBC uses to request that the database engine perform some task is called SQL (pronounced *SEE-kwel*), an acronym for structured query language. SQL was designed by IBM and later standardized as a formal database query language. This is the language that ODBC uses to interact with a database. A full discussion on SQL is beyond the scope of this book, but this chapter

covers the general concepts that most people use. More detailed information on the SQL language can be found online. Here are a few good places to start:

- Joe Casadonte's article on SQL:  
<http://www.northbound-train.com/perl/article/SQL.html>
- Ken North's articles on SQL and ODBC:  
[http://ourworld.compuserve.com/homepages/Ken\\_North/magazine.htm](http://ourworld.compuserve.com/homepages/Ken_North/magazine.htm)
- Jim Hoffman's introduction to SQL:  
<http://w3.one.net/~jhoffman/sqltut.htm>
- SQLCourse.com is another good reference:  
<http://www.sqlcourse.com/>

Before discussing how to use SQL, first you need to understand some very simple but basic SQL concepts such as delimiters and wildcards.

## Using SQL Keywords

*SQL is not case sensitive, so preserving case is not imperative; however, conventionally, SQL keywords are all in caps. So if you were to issue a query like this:*

```
SELECT * FROM Foo
```

*It would yield the same results as this:*

```
SELECT * froM Foo
```

## Delimiters

When you need to specify a separation of items, you use a delimiter. Typically, a delimiter is an object (such as a character) that symbolizes a logical separation. We all know about these, but some might not know them by name. When you refer to a path such as `c:\perl\lib\win32\odbc.pm`, for example, the backslashes (\) delimit the directory names with the last backslash delimiting the filename. The colon (:) delimits the drive letter from the directory names. In other words, each backslash indicates a new directory; backslashes separate directory names, or they *delimit* the directory names.

Many coders will delimit things to make them easier to handle. You might want to save a log file of time, status, and the name of the user who ran a script, for example. You could separate them by colons so that later you could parse them out. If you save the line `joel:896469382:success` to a file, when you read it back, you can parse it out using [Example 7.1](#).

### **Example 7.1 Parsing data using delimiters**

```
01. open( FILE, "< test.dat" ) || die "Failed to open: ($!)" ;
02. while( <FILE> )
03. {
04.     my( @List ) = split( ":", $_ );
```

```

05.     print "User=$List[0]\n";
06.     print "Date=" .. localtime( $List[1] ), "\n";
07.     print "Status=$List[2]\n";
08. }
09. close( FILE );

```

When it comes to SQL, delimiters are quite important. SQL uses delimiters to identify *literals*. A literal is just a value that you provide. If you are storing a user's name of '`Frankie`' into a database, for example, '`Frankie`' is a literal. Perl would refer to it as a value (as in, "you assigned the value

`'Frankie'` to the variable `$UserName`"). In the SQL query in following SQL query string, the number `937` and the string '`Noam Chomsky`' are both considered literals:

```
SELECT * FROM Foo WHERE Name = 'Noam Chomsky' AND ID > 937
```

When you delimit a literal, you must understand that there are actually two separate delimiters: one for the beginning of the literal (known as the *literal prefix*) and one for the end of the literal (the *literal suffix*). If that isn't enough to remember, consider this: Each data type has its own set of delimiters! Therefore, you use different delimiters when specifying a numeric literal, a time literal, a text literal, a currency literal, and the list goes on.

The good news is that it is not too difficult to discover the delimiters that a particular ODBC driver expects you to use. By using the `GetTypeInfo()` method, you can discover what delimiters, if any, are required by your ODBC driver for a particular data type (as in [Example 7.2](#)). For more on how to use this method, see the "Retrieving Data Type Information" section later in this chapter.

### **Example 7.2 Determining literal delimiters**

```

01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "MyDSN";
03. my $String = "Delimited Char String";
04. my %Type;
05. my $db = new Win32::ODBC( $DSN ) || die "Could not connect";
06. if( $db->GetTypeInfo( $db->SQL_CHAR ) )
07. {
08.     if( $db->FetchRow() )
09.     {
10.         %Type = $db->DataHash();
11.     }
12. }
13. print "Example of a properly delimited string for this data
source:\n";
14. print $Type{LITERAL_PREFIX}, $String, $Type{LITERAL_SUFFIX},
"\n";
15. $db->Close();

```

## Older Versions of Win32::ODBC

*It is important to note that the `GetTypeInfo()` method only appears in updated versions of the extension (version 19980806 and later). Currently, ActiveState does not distribute this version with its ActivePerl package. It can be downloaded from the Roth Consulting web site (<http://www.roth.net/perl/odbc/>).*

*If you have an older version of the `Win32::ODBC` extension that does not support the `GetTypeInfo()` method, you can modify the `ODBC.PM` file with the following code to enable the method:*

```
01. sub GetTypeInfo
02. {
03.     my($self, $Type) = @_;
04.     my($Connection, @Results);
05.     if(! ref $self)
06.     {
07.         $Type = $self;
08.         $self = 0;
09.         $Connection = 0;
10.    }
11.    else
12.    {
13.        $Connection = $self->{ 'connection' };
14.    }
15.    @Results = ODBCGetTypeInfo( $Connection, $Type );
16.    return ( processError( $self, @Results ) )[0];
17. }
```

The even better news is that most databases follow a simple rule of thumb: Text literals are delimited by single quotation marks (as both the prefix and suffix delimiters) and all other literals are delimited by null strings (in other words, they do not require delimiters). So you can usually specify a SQL query such as the following line, where the text literal is delimited by single quotation marks and the numeric literal has null or empty string delimiters (nothing delimiting it).

```
SELECT * FROM Foo WHERE Name = 'Noam Chomsky' AND ID >> 937
```

The use of delimiters when specifying literals can cause problems when you need to use a delimiter in the literal itself. If the prefix and suffix delimiter for a test literal is the single quotation mark ('), for example, it causes a problem when the literal text has a single quotation mark or apostrophe within itself. The problem is that because the text literal has an apostrophe (a single quotation mark), it looks to the database engine as if you are delimiting only a part of your literal, leading the database engine to consider the remaining part of the literal as a valid SQL keyword. This usually leads to a SQL error. The following statement has a text literal of 'Zoka's Coffee Shop':

```
SELECT * FROM Foo WHERE Company like 'Zoka's Coffee Shop'
```

Notice the three text delimiters, one of which is meant to be the apostrophe in `Zoka's`. This will be parsed out so that the database will think you are looking for a company name of `Zoka` and that you

are using a SQL keyword of `s Coffee Shop` and then starting another text literal without. This will result in an error:

```
$CompanyName = 'Zoka's Coffee Shop';
$Sql = "SELECT * FROM Foo WHERE Company like '$CompanyName' ";
```

The way around this is to escape the apostrophe. This is performed by prepending the apostrophe with an escape character. There is no difference between this and how Perl uses the backslash escape character when printing a new line (`"\n"`) or some other special character. The SQL escape character is the single quotation mark. "But wait a moment," you might be thinking, "the single quotation mark is typically a text literal delimiter!" Well, although that is true, if SQL finds a single quotation mark in between delimiters and the single quotation mark is followed by a valid escapable character, SQL will consider it to be an escape character.

This means that when you are using a single quotation mark in your text literals, you must escape it with another single quotation mark. The most practical way to do this is to search through your strings and replace all single quotation marks with two single quotation marks (not a double quotation mark—there is a big difference):

```
$TextLiteral =~ s/''/''/g;
```

If you wanted to correct this code, you could add a function that performs the replacement for you, as in [Example 7.3](#).

### **Example 7.3 Escaping apostrophes**

```
01. my $CompanyName = Escape( "Zoka's Coffee Shop" );
02. my $Sql = "SELECT * FROM Foo WHERE Company = '$CompanyName' ";
03.
04. sub Escape
05. {
06.     my( $String ) = @_;
07.     $String =~ s/''/''/g;
08.     return( $String );
09. }
```

### **Tip**

*Not all ODBC drivers use single quotation marks to delimit text literals. Some might use other characters, and some might go as far as to use different delimiters for the beginning and ending of a literal.*

*The way you find out which characters to use is by calling the `GetTypeInfo()` method, which is discussed in the "[Retrieving Data Type Information](#)" section later in this chapter.*

## Wildcards

SQL allows for wildcards when you are querying text literals. The two common wildcards are as follows:

- **Underscore** (`_`). Matches any one character.
- **Percent sign** (`%`). Matches any number of characters (zero or more).

These wildcards are used only with the `LIKE` predicate (discussed later in this chapter in the section titled "[SELECT Statement](#)"). For now, you need to be aware that wildcards are supported by most databases (but not all). You can use the `GetInfo()` method to determine whether your ODBC driver supports wildcards (see [Example 7.4](#)).

If you need to specify a percent sign or an underscore in your text literal and not have it interpreted as a wildcard, you will have to escape it first, just like you have to escape apostrophes. The difference between escaping apostrophes and escaping wildcards is twofold:

- Not all ODBC drivers support escaping wildcards (believe it or not).
- Usually, the escape character is the backslash (`\`), but you can change it to something else.

The first of the differences is pretty much self-explanatory; not all ODBC drivers allow support for escaping wildcards. You can check whether your ODBC driver supports wildcards and escapable wildcards by using the `GetInfo()` method, as in [Example 7.4](#).

### ***Example 7.4 Determining the wildcard escape character***

```
01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "MyDSN";
03. my $db = new Win32::ODBC( $DSN ) || die "Could not connect";
04. if( "Y" eq $$db->GetInfo( $db->SQL_LIKE_ESCAPE_CLAUSE ) )
05. {
06.     my $EscapeChar = $db->GetInfo( $db-
>SQL_SEARCH_PATTERN_ESCAPE );
07.     print "Escaping wildcards is supported.\n";
08.     print "The wildcard escape character is: $EscapeChar\n";
09. }
10. else
11. {
12.     print "The data source does not support escaping
wildcards.\n";
13. }
14. $db->Close();
```

If your ODBC driver supports the use of escaped wildcards, you can set the escape character to be whatever you want it to be (you don't have to settle for what the ODBC driver uses by default) by using a `LIKE predicate escape character sequence`. This process is described later in this chapter in the section on escape clauses.

## Quoted Identifiers

At times, you might need to use special characters in a query. Assume, for example, that you have a table name with a space in its name, such as "Accounts Receivable." If you were to use the table name in a SQL query, the parser would think you are identifying two separate tables: Accounts and Receivable. This is because the space is a special character in SQL. Ideally, you would rename your table to "Accounts\_Receivable" or something else that does not have such special characters. Because this is not an ideal world all the time, ODBC provides a way around this called the *identifier quote character*.

Most databases use a double quotation mark ("") as the identifier quote character, but because some do not, you might want to query your ODBC driver to discover which character you should use. This is done using the `GetInfo()` method, which is discussed in the "[Obtaining General ODBC Information](#)" section later in this chapter.

When you need to use special characters in an identifier, surround the identifier with the identifier quote character. The following code line shows a SQL query making use of the identifier quote characters for the table name "Accounts Receivable."

```
SELECT * FROM "Accounts Receivable"
```

## SQL Grammar

When you construct a SQL statement, you are using a standardized language that has been around for a while. Dozens of good books are available that explore SQL and can provide insight on how to optimize your queries and take full advantage of SQL's powerful features. This section is just provided for the sake of completeness because SQL is the language that ODBC uses.

When you use SQL, you construct what is called a *statement*. A statement is a type of command that can have many details. If you want to get all the rows from a database table that meet some criteria (such as a value of 25 in the Age column), for example, you use a `SELECT` statement.

### **SELECT Statement**

A `SELECT` statement retrieves a set of data rows and columns (known as a dataset) from the database. It is quite simple to use. The basic structure of a `SELECT` statement is

```
SELECT [ALL | DISTINCT] columnname [, columnname]
FROM tablename
[WHERE condition]
[ORDER BY columnname [ASC|DESC] [, columnname [ASC | DESC]] -]
```

where

- `columnname` is the name of a column in the table.
- `tablename` is the name of a table in the database.
- `condition` is a condition that determines whether to include a row.

A `SELECT` statement will retrieve a dataset consisting of the specified columns from the specified table. If an asterisk (\*) is specified rather than a column list, all columns are retrieved.

Every row in the table that meets the criteria of the `WHERE` clause will be included in the dataset. If no `WHERE` clause is specified, all rows will be included in the dataset.

The rows in the dataset will be sorted by the column names listed in the order listed if you specify an `ORDER BY` clause. If you use `ORDER BY LastName, FirstName`, the resulting dataset will be sorted in ascending order (the default) first by `LastName` and then by `FirstName`.

For each column specified, you can use the `ASC` or `DESC` keyword to indicate sorting the column by ascending or descending order, respectively. If neither keyword is specified, the column will be sorted however the previous column was. If no keyword is specified for the entire `ORDER BY` clause, `ASC` will be assumed.

It is interesting to note that instead of specifying a column name for the `ORDER BY` clause, you can refer to a column number (for example, `ORDER BY 5 ASC, LastName DESC, 4, 3`).

The following code line retrieves a dataset consisting of all the fields (the asterisk indicates all fields) from all the records contained in the table called `Foo`.

```
SELECT * FROM Foo
```

The `WHERE` predicate specifies a search condition. Only rows that meet the criteria set by the `WHERE` clause are retrieved as part of the dataset. `WHERE` clauses consist of conditional statements that equate to either true or false such as `Age > 21`. [Table 7.1](#) lists the valid conditions.

**Table 7.1. Valid Conditional Predicates**

Operator	Description
<code>X = Y</code>	X is equal to Y.
<code>X &gt; Y</code>	X is greater than Y.
<code>X &lt; Y</code>	X is less than Y.
<code>X &gt;= Y</code>	X is greater or equal to Y.
<code>X &lt;= Y</code>	X is less than or equal to Y.
<code>X &lt;&gt; Y</code>	X is not equal to Y.
<code>X [NOT] BETWEEN Y AND Z</code>	X is between the values of Y and Z. If the <code>NOT</code> keyword is used, X is <i>not</i> between the values Y and Z.
<code>X [NOT] LIKE 'Y'</code>	X is the same as Y (when comparing text data types). If the <code>NOT</code> keyword is used, X is <i>not</i> the same as Y. The Y value can consist of a text literal with wildcards. This comparison is only needed when using wildcards; otherwise, it is more efficient to use =.
<code>EXISTS (subquery)</code>	This will be true for every row returned from a subquery (another <code>SELECT</code>

**Table 7.1. Valid Conditional Predicates**

Operator	Description
X IS [NOT] NULL	statement).
X [NOT] IN (Y, Z, ...)	The column X is <code>NULL</code> . If the <code>NOT</code> keyword is used, column X is <i>not</i> <code>NULL</code> .
X <operator> {ALL ANY} (subquery)	The value X is found in the list of (Y, Z, ...). If the <code>NOT</code> keyword is used, the value X is <i>not</i> found in the list (Y, Z, ...). The list (Y, Z, ...) could be another <code>SELECT</code> subquery.
X <operator> {ALL ANY} (subquery)	A subquery (another <code>SELECT</code> statement) is performed and the rows from the resulting dataset are compared to the value X using an operator, which could be any in this list. If the <code>ALL</code> keyword is used, all rows from the subquery must match the condition set by the operator. If the <code>ANY</code> keyword is used, any row that matches the condition set by the operator will return a true value.

Usually, the conditional statement includes the name of a column. When a column is referred to, you are referring to the value in the column. In [Example 7.5](#), a search condition of `Age > 21` is used. Every row that has a value greater than `21` in the `Age` column will satisfy the condition; therefore it will be returned in the dataset.

### **Example 7.5 Specifying a WHERE clause**

```
01. SELECT *
02. FROM FOO
03. WHERE Age > 21
```

[Example 7.6](#) makes use of the `WHERE` predicate to indicate a condition of the search. The query will retrieve a dataset consisting of the `Firstname`, `Lastname`, and `City` fields in the `Users` table only if the user's first name begins with the letter `C` (the `%` is a wildcard that indicates that you don't care what comes after the `C`; refer to the section on wildcards). The dataset will then be sorted by `Lastname` (by default, the order will be ascending).

### **Example 7.6 Complex SELECT query**

```
01. SELECT Firstname, Lastname, City
02. FROM Users
03. WHERE Firstname like "C%"
04. ORDER BY Lastname
```

Multiple conditions can be used in a `WHERE` clause by using Boolean conjunctions such as AND and OR. [Example 7.7](#) will return a dataset with rows whose `Age` column has a value of greater than `21` and whose `Lastname` column begins with either the letter `C` or `R`.

### **Example 7.7 Using multiple conditions in a SELECT statement**

```
01. SELECT *
02. FROM FOO
```

```
03. WHERE Age > 21 AND  
04.      (LastName like 'C%' OR LastName like 'R%')
```

For another `SELECT` statement, consider [Example 7.8](#). This query will return a dataset consisting of all fields only if the `City` field is `Seattle` and the `Zip` code is not equal to `98103`. The list will then be sorted in descending order (starting with Zs and ending with As) by last name.

### **Example 7.8 SELECT statement with sorting and a condition**

```
01. SELECT *  
02. FROM Users  
03. WHERE City like 'Seattle' AND Zip <> 98103  
04. ORDER BY Lastname DESC
```

## ***INSERT Statement***

An `INSERT` statement adds a row to a table in a database. The structure of an `INSERT` statement is

```
INSERT INTO tablename  
[(columnname1[, columnname2] ... )]  
VALUES (value1[, value2] ... )
```

where

- `tablename` is the name of a table in the database.
- `columnname` is the name of the column to receive a value.
- `value` is the value to be placed into the column.

The `INSERT` statement is used when you need to add a record to a table. The trick here is that when you specify a list of column names, you must list the values for those columns in the same order as the columns.

The list of column names is optional, and if left out, the first value specified will be stored in the first column, the second value will be stored in the second column, and so on.

The following code adds a row into the table `Foo` and assigns the value `Dave` to the `Name` column, `Seattle` to the `City` column, and `98103` to the `Zip` column.

```
INSERT INTO Foo  
(Name, City, Zip)  
VALUES ('Dave', 'Seattle', 98103)
```

Just to demonstrate how you can use the `INSERT` statement without providing any column names, look at the following:

```
INSERT INTO Foo  
VALUES ('Zoka''s Coffee Shop', 'Double Late', 2.20)
```

This example assumes that the first and second columns are text data types and that the third is either a floating type or a currency type (notice that the example escapes the apostrophe in the first value).

## Note

*It is rather important to understand that although the SQL language is fairly standardized, not all data sources implement it in the same way. To MS SQL Server, the `INTO` keyword for an `INSERT` statement is optional, for example; however, MS Access requires it.*

*It is most practical to never use data source-specific shortcuts or leave out optional keywords unless you are sure your script will always be used with a particular data source.*

## UPDATE Statement

Not only can you select and insert data into a table, you can also change the values in a given row of a table. The capability to *update* a row is tremendously powerful and is performed using an `UPDATE` statement. The statement's syntax is similar to that of the `INSERT` statement:

```
UPDATE tablename  
SET columnname1=value1  
    [, columnname2=value2]  
...  
[WHERE condition]
```

where

- `tablename` is the name of a table in the database.
- `columnname` is the name of the column to receive a value.
- `value` is the value to which the column should be set.
- `condition` is a search condition as defined in the `SELECT` statement section.

This statement can be used to modify the existing rows of data. The following code uses an `UPDATE` statement to change the `Department` column for all rows in the `Users` table. If a row's `Department` column contains the value `Personnel`, it is changed to the more politically correct `HumanResources`.

```
UPDATE Users  
SET Department = 'Human Resources'  
WHERE Department = 'Personnel'
```

## DELETE Statement

Removing a row from a table is not difficult at all. This is done with a `DELETE` statement:

```
DELETE [FROM] tablename  
[WHERE condition]
```

where

- `tablename` is the name of a table in the database from which rows will be deleted.
- `condition` is a search condition as defined in the `SELECT` statement section.

The first `FROM` keyword is optional—some data sources might require it. The `tablename` (following the optional first `FROM` keyword) is the table that will be affected by the statement.

If search condition specified is `true` for a row, the row is deleted from `tablename`. In the following code, all rows from the table `Students` will be deleted as long as the row has a value greater than `4` in the `Year` column and the `SchoolName` column contains the value `Michigan State University`.

```
DELETE FROM Students  
WHERE Year > 4 AND  
SchoolName = "Michigan State University"
```

If no search condition is specified, all rows from `tablename` are deleted, as in the following:

```
DELETE FROM Students
```

## Escape Sequences

Each and every data source has its own way of dealing with specific data types. The `DATE` data type format for Oracle 6 is `Aug 20, 1997`, for example, but IBM's DB2 is `1997-08-20`. This becomes quite a problem when creating a SQL statement in which you have no idea what the actual database engine will be.

Because ODBC attempts to abstract the particulars of different databases, the ODBC standard has adopted a technique to contend with this problem. This technique is called an *escape sequence*. When an ODBC driver finds an escape clause in a SQL statement, it converts it to whatever format it needs to be to suit the particular database to which the statement will be sent.

## Date and Times

The escape clause for a date is `{d 'yyyy-mm-dd'}`. To create an ODBC SQL statement that will be properly interpreted by all databases, you could use the following:

```
SELECT * FROM Foo WHERE Date = {d '1997-08-20'}
```

The time/date-based escape sequences are as follows:

- **Date:** `{d 'yyyy-mm-dd'}`
- **Time:** `{t 'hh:mm:ss'}`
- **Timestamp:** `{ts 'yyyy-mm-dd hh:mm:ss'}`

## Outer Joins

Often, it is useful to link two tables together. For example, let's say you have a table with book names and their author names. Let's also say you have another table with author names and their phone numbers. It would be great if you could *join* these two different tables so that, by looking up a book, you automatically have access to the author's phone number. This is where the SQL *join* statement comes in. It enables a SQL query to link multiple tables together. There are multiple types of joins such as outer, inner, and equijoins. Any good SQL reference will describe the use of joins because they are very powerful and useful. You can find a good online reference to explaining joins at <http://sqlcourse2.com/joins.html>.

ODBC defines an escape clause for a flavor of joins called the *outer join*. If you need to use an outer join, you can use the escape sequence

```
{oj outer_join}
```

where the *outer join* consists of

```
tablename {LEFT | RIGHT | FULL} OUTER JOIN{tablename | outer_join}  
ON  
search_condition
```

In the following code, the outer join specifies all the fields from every record of the `Machine` table and every record from the `Users` table in which the field `MachineName` (from the `Users` table) matches `Name` (from the `Machine` table), as long as the field `Processor` (from the `Machine` table) is greater than 486.

```
SELECT *  
FROM {oj Machine LEFT OUTER JOIN Users ON Machine.Name =  
Users.MachineName}  
WHERE Machine.Processor > 486
```

Because this query uses an outer join as an escape sequence, you can be guaranteed that it will work on any ODBC driver that supports outer joins. Even if the particular driver uses a nonstandard syntax for outer joins, the ODBC driver will convert the escape sequence into the correct syntax before executing it.

Before you actually make use of an outer join, you might want to check to make sure your ODBC driver supports them. [Example 7.9](#) shows a simple line that will check for outer join support.

### **Example 7.9 Discovering whether an ODBC driver supports outer joins**

```
01. use Win32::ODBC;  
02. my $DSN = shift @ARGV || "MyDSN";  
03. my $db = new Win32::ODBC( $DSN ) || die "Could not connect";  
04. my $Capabilities = $db->GetInfo( $db->SQL_OJ_CAPABILITIES );  
05. if( $db->SQL_OJ_FULL & $Capabilities )  
06. {  
07.   print "This data source supports full outer joins\n";
```

```

08. }
09. else
10. {
11.   print "This data source does not support full outer
joins\n";
12. }
13. $db->Close();

```

## Scalar Functions

Scalar functions, such as time and date, character, and numeric functions, can be implemented by means of escape sequences

```
{fn function}
```

where function is a scalar function supported by the ODBC driver. The following code compares the `Date` column (which is of a timestamp data type) with the current date.

```

SELECT *
FROM Foo
WHERE Date = {fn curdate()}

```

For a full list of scalar functions, see Appendix B, "Win32::ODBC Specific Tables" (Appendix B is available on this book's Web site). Before using a scalar function, you should see whether it is supported by the ODBC driver.

You can use [Example 7.10](#) to discover this. The value returned from `GetInfo( $db->SQL_TIMEDATE_FUNCTIONS )` is a bitmask for all the supported time and date functions.

### **Example 7.10 Checking whether the ODBC driver supports the `curdate()` function**

```

01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "MyDSN";
03. my $db = new Win32::ODBC( $DSN ) || die "Could not connect";
04. my $Capabilities = $db->GetInfo( $db->SQL_TIMEDATE_FUNCTIONS );
05. if( $db->SQL_FN_TD_CURDATE & $Capabilities )
06. {
07.   print "This data source supports the curdate() function.\n";
08. }
09. else
10. {
11.   print "This data source does not support the curdate()
function.\n";
12. }
13. $db->Close();

```

## Stored Procedures

Stored procedures can be called by means of escape sequences. The syntax is as follows:

```
{[?=]call procedure name([parameter][,parameter]...])}
```

If a return value is expected, you need to specify the preceding `?=`, as in the following:

```
{? = call MyFunction('value')}
```

Otherwise, a call can be constructed like this:

```
{call MyFunction('value')}
```

### Note

*Older versions of Win32::ODBC do not support parameter binding, so using the `?` as a "passed in" parameter to a stored procedure is not supported and will probably result in an error.*

*The only exception to this is the return value of a stored procedure. For example, if a SQL statement were submitted like this:*

```
{? = call MyStoredProc( 'value1', value2 ) }
```

*The value returned by the procedure would be stored in a dataset containing one row. Future versions of this extension will support parameter binding.*

## LIKE Predicate Escape Character

Even though it is not common to do so (because you can always use the default escape character), you can change the character used to escape a wildcard by using the `LIKE` predicate escape character sequence

```
{escape 'character'}
```

where `character` is any valid SQL character.

The following code demonstrates this:

```
SELECT *
FROM Foo
WHERE Discount LIKE '28#%' {escape '#'}
```

This example will return all columns from the table `FOO` that have `28%` in the `Discount` column. Notice that by specifying the `LIKE` predicate escape character sequence, the pound character (#) is used to escape the wildcard. This way, the wildcard is deemed a regular character and not interpreted as a wildcard. Because the resulting condition will not include any wildcards, the condition could have been this:

```
WHERE Discount = '28%'
```

This particular example, however, demonstrates the use of the `LIKE` predicate escape clause, so we are using the `LIKE` predicate.

Usually, this escape sequence is not necessary because the default escape character is the backslash (\) and the same SQL statement could have been this:

```
SELECT *
FROM FOO
WHERE Discount LIKE '28\%'
```

## How to Use `Win32::ODBC`

The `Win32::ODBC` extension attempts to make connecting to an ODBC data source as easy as possible. In doing this, it hides many details from the user. This is either a good or bad thing, depending on your point of view. Nonetheless, the extension is designed to be as flexible as the ODBC API is, without requiring the user to learn the API itself. Of course, to utilize the more exotic ODBC features, a good book on ODBC is recommended; otherwise, you will drive yourself insane with all the constants, functions, and terminology!

When an application wants to connect to an ODBC data source, it must create the following items in order:

1. An ODBC environment
2. An ODBC connection to the data source
3. A statement handle

Because all database interaction uses a statement handle, all three steps must be performed, in order, before any database interaction can occur. It is possible to have multiple statement handles per connection. This is how a script can have multiple database queries open simultaneously.

`Win32::ODBC` attempts to simplify this process. When you create a new `Win32::ODBC` object, all interaction with the data source goes through this object. If you need to connect to many data sources at once, you just create many `Win32::ODBC` objects. For those users who need to create multiple queries to one data source (the equivalent of having multiple statement handles), the capability to *clone* an object has been implemented. Cloned objects share the same ODBC connection but have separate statement handles.

The use of `Win32::ODBC` is very straightforward in most instances; there are basically five steps:

1. Connecting to the database
2. Submitting a query

3. Processing the results
4. Retrieving the data
5. Closing the database

Because `Win32::ODBC` is a Perl extension, it must be loaded by means of the `use` command. Typically, this is placed in the beginning of your script, as in the following:

```
use Win32::ODBC;
```

## Connecting to the Database

After your Perl application loads the `Win32::ODBC` extension, it needs to initiate a conversation with a database. This is performed by creating an ODBC connection object using the `new` command:

```
$db = new Win32::ODBC( $DSN [, $Option1, $Option2, ... ] );
```

The first parameter (`$DSN`) is the data source name. This parameter can be either a DSN or a DSN string. This is described later in this section.

The optional additional parameters are connection options that can also be set by using the `SetConnectOption()` method. Some options, however, must be set before the actual connection is made to the ODBC driver and database. Therefore, you can specify them in the `new` command.

**Example 7.11** assumes a few things: It assumes that you have already created a DSN called "My DSN" and that it is configured correctly. The example also assumes that the current user has permissions to access the DSN. This is not so much a problem for Windows 95 users as it is for Windows NT users.

### **Example 7.11 Connecting to a DSN**

```
use Win32::ODBC;
my $db = new Win32::ODBC( "MyDSN" );
```

If all went well, you will have an object, `$db`, that you will use later. Otherwise, something went wrong, and the attempt to connect to the database failed. If something did go wrong, the object will be empty. A simple test, as in [Example 7.12](#), can be used to determine success or failure in connecting to the database. If the connection fails, the script will die printing out the error. Error processing is discussed later in this chapter in the section titled, strangely enough, "[Error Processing](#)."

### **Example 7.12 Testing whether a connection to a database succeeded**

```
01. use Win32::ODBC;
02. my $db = new Win32::ODBC( "MyDSN" );
03. if( ! $db )
04. {
05.   die "Error connecting: " . Win32::ODBC::Error() . "\n";
06. }
07. $db->Close();
```

You can override your DSN configuration by specifying particular configuration elements in the DSN name when creating a new ODBC connection object. To do this, your DSN name must consist of driver-specific keywords and their new values in the following form:

```
keyword=value;
```

You can specify as many keywords as you like and in any order you like, but the "DSN" keyword must be included (and should be the first one). This keyword indicates which DSN you are using and the other keyword/value pairs that will override the DSN's configuration.

[Table 7.2](#) provides a list of standard keywords. A data source might allow additional keywords to be used (for example, MS Access defines the keyword `DBQ` to represent the path to the database file).

**Table 7.2. Standard DSN Keywords**

Keyword	Description
<code>DSN</code>	The value would be the name of an existing DSN.
<code>FILEDSN</code>	The value would be the name of a file-based DSN. Such files end with the <code>.DSN</code> extension. <i>This is only available with ODBC 3.0 or higher.</i>
<code>DRIVER</code>	A description of the ODBC driver to be used. This value can be obtained by using the <code>Win32::ODBC::Drivers()</code> function.
<code>UID</code>	The value represents a user ID (or username) that will be sent to the data source.
<code>PWD</code>	The value represents a password that will be sent to the data source.
<code>SAVEFILE</code>	The value is a name of a file-based DSN that the DSN string will be saved as. <i>This is only available with ODBC 3.0 or higher.</i>

## Note

If ODBC 3.0 or higher is being used, there are two keywords that cannot be used together: `DSN` and `FILEDSN`. If they both appear in a connection string, only the first one is used.

Additionally, ODBC 3.0 allows for a DSNless connection string in which no `DSN` or `FILEDSN` keyword is used. A `DRIVER` keyword must be defined, however, in addition to any other keywords necessary to complete the connection.

Suppose a DSN exists called `OZ` that points to an Access database. If you want to use that DSN but specify a user ID to log on to the database as (in Access, a keyword of `UID`) and a password (keyword of `PWD`), you could use the following line:

```
$db = new Win32::ODBC( "DSN=OZ;UID=dorothy;PWD=noplaceIlikehome" );
```

A connection would be made to the `oz` DSN, but overriding the default user ID and password would be difficult.

Other than the practical limitations of memory, you have no real limit as to how many database connections you can have. Although some tricks can help speed things up, you should conserve memory use and make the most of an ODBC connection (more on that later).

After you have your ODBC object(s), you are ready to begin querying your database.

### Tip

*The `Win32::ODBC` extension's functions also map to another namespace called `ODBC`. It is not necessary to always use the full `Win32::ODBC` namespace when calling functions or constants. For example, the following two function calls are the same:*

```
@Error = Win32::ODBC::Error();
@Error = ODBC::Error();
```

*Likewise, when you create a new ODBC object, you can use either of the following:*

```
$db = new Win32::ODBC( "MyDSN" );
$db = new ODBC( "MyDSN" );
```

*Anywhere you might need to use the full namespace of the extension, you instead use the abbreviated version: `ODBC`.*

## Submitting a Query

Now that you have an ODBC connection object, you can begin to interact with your database. You need to submit a SQL query of some sort. This is where you use the `Sql()` method:

```
$db->Sql( $SqlStatement );
```

The first and only parameter is a SQL statement. This can be any valid SQL statement.

The `Sql()` method will return a nonzero integer corresponding to a SQL error number if it is unsuccessful.

### Note

*It is very important to note that the `Sql()` method is the only method that returns a nonzero integer upon failure.*

*The reason for this is to remain backward compatible with the original version of `Win32::ODBC`, then called `NT::ODBC`, which was written by Dan DeMaggio.*

*Originally, NT::ODBC used the return value of the `sql()` method (then the method was all lowercase) to indicate the error number. This technique of error checking has been made obsolete with the introduction of the `Error()` method. For the sake of backward compatibility, however, the return values have not changed.*

Suppose you have a database called `oz` with a table called `Characters`. The table consists of the fields (also known as columns) in [Table 7.3](#).

**Table 7.3. The Table Called Characters in a Fictitious Database Called OZ**

Field Name	Data Type
Name	<code>char(20)</code>
Origin	<code>char(20)</code>
Goal	<code>char(30)</code>

Suppose also that you want to query the database and find out who is in `oz`. [Example 7.13](#) will connect to the database and submit the query.

#### **Example 7.13 Submitting the SQL query**

```
01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "OZ";
03. my $Table = shift @ARGV || "Characters";
04. my $db = new Win32::ODBC( $DSN );
05. if( ! $db )
06. {
07.     die "Error connecting: " . Win32::ODBC::Error() . "\n";
08. }
09. if( $db->Sql( "SELECT * FROM $Table" ) )
10. {
11.     print "Error submitting SQL statement: " . $db->Error() .
"\n";
12. }
13. else
14. {
15.     #... process data ...
16. }
17. $db->Close();
```

The SQL statement `SELECT * FROM $Table` in line 9 will, if successful, return a dataset containing all fields from all rows of the database. By passing this statement into the `Sql()` method, you are requesting that the database perform this query.

If the `Sql()` method returns a nonzero result, there was an error and the error is printed. If the query was successful (a return value of zero), you will need to move on and process the results.

## Processing the Results

After you have a dataset that has been prepared by a SQL statement, you are ready to process the data. The way this is achieved is by moving, row by row, through the dataset and extracting the data from columns in each row.

The first thing you must do is tell the ODBC connection that you want to move to the next available row. Because you just performed the query, you are not even looking at a row yet; therefore, you need to move to the next available row, which in this case will be the first row. The method used to move from row to row is `FetchRow()` :

```
( $Result, @RowResults ) = $db->FetchRow( [ $Row[ , $Mode ] ] );
```

Before explaining the optional parameters, it is important to understand that the parameters usually are not used. They refer to the extended capabilities the ODBC function `SQLExtendedFetch()`.

The first optional parameter (`$Row`) is the row number to which you want to move. For more details, see the aforementioned section on advanced `Win32::ODBC` features.

The second optional parameter (`$Mode`) is the mode in which the cursor will be moved. If this parameter is not specified, `SQL_FETCH_RELATIVE` mode is assumed. For more details, see the aforementioned section on advanced `Win32::ODBC` features.

The `FetchRow()` method will return a of 1 if it successfully moved to the next row. If it returns a zero, it usually means that no more data is left (that is, you have reached the end of your dataset) or that an error has occurred.

Another value is returned but is typically not of any use unless you use the advanced features of `FetchRow()`. For more information, see the section "Advanced Row Fetching" later in the chapter.

[Example 7.14](#) shows a typical usage of the `FetchRow()` method. The loop continues (lines 16 through 20) to execute as long as you can advance to the next row. After the last row has been obtained, `$db->FetchRow()` returns a `FALSE` value that causes the loop to terminate.

### **Example 7.14 Fetching rows**

```
01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "OZ";
03. my $Table = shift @ARGV || "Characters";
04. my $db = new Win32::ODBC( $DSN );
05. if( ! $db )
06. {
07.   die "Error connecting: " . Win32::ODBC::Error() . "\n";
08. }
09. if( $db->Sql( "SELECT * FROM $Table" ) )
10. {
11.   print "Error submitting SQL statement: " . $db->Error() .
"\n";
12. }
13. else
14. {
```

```

15. my $iCount = 0;
16. while( $db->FetchRow() )
17. {
18.     printf( "Row %d\n", ++$iCount );
19.     #... process data ...
20. }
21. }
22. $db->Close();

```

## Note

*As of version 970208, `FetchRow()` makes use of the `SQLExtendedFetch()` ODBC function. Unfortunately, not all ODBC drivers support this and therefore fail when `FetchRow()` is called.*

*If this happens, you can either use another ODBC driver, such as a newer version or one from another vendor, or you can use an older version of `Win32::ODBC`. Versions of the extension before 970208 use the regular `SQLFetch()` ODBC function. Future releases of `Win32::ODBC` will support both fetch methods and will use whichever one the ODBC driver supports.*

## Retrieving the Data

After a row has been fetched, the data will need to be retrieved from it. There are two methods for doing this: the `Data()` method and the `DataHash()` method.

The `Data()` method returns an array of values:

```
@Data = $db->Data( [$ColumnName1[, $ColumnName2 ... ]] );
```

A list of parameters can be passed in. Each parameter is a column name from which you are retrieving data. The values passed in are case sensitive.

The `Data()` method returns an array of values corresponding, in order, to the column names passed into the method. If nothing is passed into the method, all column values are returned in the order in which they appear in the row. [Example 7.15](#) shows how the `Data()` method can be used (lines 18 and 19). Note that the parameters passed into the `Data()` method in line 18 are case sensitive.

### Example 7.15 Using the `Data()` method

```

01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "OZ";
03. my $Table = shift @ARGV || "Characters";
04. my $db = new Win32::ODBC( $DSN );
05. if( ! $db )
06. {
07.     die "Error connecting: " . Win32::ODBC::Error() . "\n";
08. }
09. if( $db->Sql( "SELECT * FROM $Table" ) )
10. {

```

```

11.    print "Error submitting SQL statement: " . $db->Error() .
12.    "\n";
13. }
14. {
15.    my $iCount = 0;
16.    while( $db->FetchRow() )
17.    {
18.        my @Data = $db->Data( "Name", "Origin" );
19.        printf( "Row %d: %s from %s\n", ++$iCount, $Data[0],
20.                $Data[1] );
21.    }
22. $db->Close();

```

Notice how the first element in the `@Data` array (`Data[0]`) is the value of the first column name specified. The order of the column names passed into the `Data()` method (line 18) determines the order in which they are stored in the array.

The second and more practical way to retrieve data is to use the `DataHash()` method. The `DataHash()` method is the preferred method when retrieving data because it associates the column name with the column's data. The `DataHash()` method returns either an `undef` if the method failed or a hash with column names as keys and the hash's values consisting of the column's data:

```
%Data = $db->DataHash( [ $ColumnName1[, $ColumnName2 ... ] ] );
```

A list of parameters can be passed in. Each parameter is a column name from which you are retrieving data. If no parameters are passed in, the data for all columns will be retrieved.

If you were to query the table previously described in [Table 7.3](#) using the code in [Example 7.16](#), you might get output that looks like this:

```
Dorothy is from Kansas and wants to go home.  
The Scarecrow is from Oz and wants a brain.  
The Wicked Witch is from The West and wants the ruby slippers.
```

This output would continue until all rows have been processed.

### **Example 7.16 Extracting data using the `DataHash()` method**

```

01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "OZ";
03. my $Table = shift @ARGV || "Characters";
04. my $db = new Win32::ODBC( $DSN );
05. if( ! $db )
06. {
07.     die "Error connecting: " . Win32::ODBC::Error() . "\n";
08. }
09. if( $db->Sql( "SELECT * FROM $Table" ) )

```

```

10. {
11.   print "Error submitting SQL statement: " . $db->Error() .
12. }
13. else
14. {
15.   my $iCount = 0;
16.   while( $db->FetchRow() )
17.   {
18.     my %Data;
19.     %Data = $db->DataHash();
20.     printf( "%d) %s is from %s and wants %s.\n",
21.             ++$iCount,
22.             $Data{Name},
23.             $Data{Origin},
24.             $Data{Goal} );
25.   }
26. }
27. $db->Close();

```

## Tip

*When using the `DataHash()` method, it is best that you `undef` the hash used to hold the data before the method call. Because the `DataHash()` method is typically called from within a while loop that fetches a row and retrieves data, it is possible that the hash might retain values from a previous call to `DataHash()`*

## Closing the Database

So far, you have opened the database, submitted a query, and retrieved and processed the results. Now you need to finish by closing the database. The proper way to perform this is by calling the `Close()` method. This method tells the ODBC Manager to properly shut down the ODBC connection. The ODBC Manager will, in turn, tell the driver to clean up after itself (removing temporary files, closing network connections, flushing buffers, and so on), and the ODBC connection object will be destroyed so that it cannot be used any longer. Here is the syntax for the `Close()` method:

```
$db->Close();
```

The `close()` method has no return value.

[Example 7.16](#) in the preceding section shows a full working example of how you might use the `Win32::ODBC` module, including the `Close()` method.

# The Win32::ODBC API

Win32::ODBC supports a rich set of features, most of which are never used (yet still exist). Some of these require knowledge of ODBC that is beyond the scope of this book. You can find several books that do discuss this topic.

## Constants

For almost every method and function in Win32::ODBC, a set of constants is needed. These constants represent a numeric value that might change as the ODBC API changes over time, so it is important for you to use the constant names and not the values they represent.

One of the most questioned aspects of the Win32::ODBC extension is how to make use of the constants. Only a small group of constants are actually exported from the ODBC module, so to use most of the constants, you need to specify a namespace such as the following:

```
Win32::ODBC::SQL_COLUMN_TABLE_NAME
```

Or, if you have an ODBC connection object, you can access the constant value as if it were a member of the object, as follows:

```
$db->SQL_DATA_SOURCE_NAME
```

Because Win32::ODBC creates a synonymous ODBC namespace and maps it to Win32::ODBC, you could use this:

```
ODBC::SQL_CURSOR_COMMIT_BEHAVIOR
```

Notice that the preceding example just uses the ODBC namespace rather than the Win32::ODBC namespace. Both are valid, but the latter is a bit shorter.

The reason all constants are not exported into the main namespace is because the ODBC API defines more than 650 constants, each of which is important to have. The decision was made to not export all the constants because it would bloat memory use and clutter the main namespace with an entire list of constants that will most likely not be used.

You could always edit the ODBC.pm file and export constants that you would like to export, but then again, it is not so difficult to just make use of one of the formats listed earlier.

## Catalog Functions

The ODBC API supports metadata functions, such as cataloging. Win32::ODBC supports access to such information with two methods:

```
$db->Catalog( $Qualifier, $Owner, $Name, $Type );
$db->TableList( $Qualifier, $Owner, $Name, $Type );
```

Both of these are really the same method, so they can be used interchangeably. The only difference is in how the results are obtained.

The first parameter (`$Qualifier`) represents the source the database will use. In Access, for example, the `$Qualifier` would be the database file (such as `c:\data\mydatabase.mdb`); in MS SQL Server, it would be the database name.

The second parameter (`$Owner`) is the owner of the table. Some database engines can put security on tables either granting or denying access to particular users. This value would indicate a particular owner of a table—that is to say, the user either who created the table or to whom the table has been given.

The third parameter (`$Name`) is the name of the table.

The fourth parameter (`$Type`) is the table type. This can be any number of database-specific types or one of the following values:

- TABLE
- VIEW
- SYSTEM TABLE
- GLOBAL TEMPORARY
- LOCAL TEMPORARY
- ALIAS
- SYNONYM

A call to the `Catalog()` or `TableList()` method can include search wildcards for any of the parameters. Passing in `USER%` for the third parameter will result in retrieving all the tables with names that begin with `USER`.

The difference between these two methods is that the `Catalog()` method returns a result set that you need to process with `FetchRow()` and `Data()` or `DataHash()`. On the other hand, `TableList()` returns an array of table names that meet the criteria you specified.

Basically, `TableList()` is a quick way of getting a list of tables, and `Catalog()` is a way of getting much more information.

If the `Catalog()` method is successful, it returns `TRUE` and results in a dataset with four columns: `TABLE_QUALIFIER`, `TABLE_OWNER`, `TABLE_NAME`, `TABLE_TYPE`, and `REMARKS`. There might also be additional columns that are specific to the data source. Each row will represent a different table. You can use the normal `FetchRow()` and `DataHash()` methods to retrieve this data. If the method fails, `FALSE` is returned.

## Note

*The resulting dataset generated by a call to the `Catalog()` method will produce a different result set if ODBC 3.0 or higher is used. In this case, `TABLE_QUALIFIER` becomes `TABLE_CAT` and `TABLE_OWNER` becomes `TABLE_SCHEM`. This is due to a change in the ODBC specification for version 3.0.*

If the `TableList()` method is successful, it will return an array of table names.

If all parameters are empty strings except the fourth parameter (table type), which is only a percent sign (%), the resulting dataset will contain all valid table types. You can use this when you need to discover which table types a data source can use.

If the `$Owner` parameter is a single percent sign and the qualifier and name parameters are empty strings, the resulting dataset will contain the list of valid owners that the data source recognizes.

If the `$Qualifier` is only a percent sign and the `$Owner` and `$Name` parameters are empty strings, the resulting dataset will contain a list of valid qualifiers. [Example 7.17](#) describes how you can use both of these methods.

### **Example 7.17 Using the `TableList()` and `Catalog()` methods**

```
01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "OZ";
03. my $Table = shift @ARGV || "Characters";
04. my $TableType = "'TABLE','VIEW','SYSTEM TABLE','".
05.                 "'GLOBAL TEMPORARY','LOCAL TEMPORARY','".
06.                 "'ALIAS','SYNONYM'";
07. my $db = new Win32::ODBC( $DSN );
08. if( ! $db )
09. {
10.     die "Error connecting: " . Win32::ODBC::Error() . "\n";
11. }
12. my $iCount = 0;
13. print "List of tables:\n";
14. map{++$iCount; print "$iCount) $_\n";} $db->TableList();
15.
16. print "\nFull list of tables:\n";
17. if( $db->Catalog( "", "", "%", $TableType ) )
18. {
19.     $iCount = 0;
20.     while( $db->FetchRow() )
21.     {
22.         my %Data = $db->DataHash();
23.         print ++$iCount, " "
${Data{TABLE_NAME}}\t${Data{TABLE_TYPE}}\n";
24.     }
25. }
26. else
27. {
28.     print "Error submitting SQL statement: " . $db->Error() .
"\n";
29. }
30. $db->Close();
```

## Managing Columns

Each column (or field—whichever way you prefer to say it) can be of a different data type. One column might be a text data type (for a user's name, for example), and another might be a numeric data type representing a user's age.

At times, a programmer might need to learn about a column, things such as what data type the column is, whether the programmer can conduct a search on that column, or whether the column is read-only. If the programmer created the database table, chances are he already knows this information; if the table came secondhand, the programmer might not know such information. This is where the `ColAttributes()` method comes into play:

```
$db->ColAttributes( $Attribute, [ @@ColumnNames ] );
```

The first parameter (`$Attribute`) is the attribute, a numeric value representing attributes of a column. Appendix B, "Win32:ODBC Specific Tables," contains a list of valid constants that can be used. (Appendix B is on the web site for this book at [www.roth.net/books/extensions](http://www.roth.net/books/extensions).)

Additional parameters might be included—specifically, the names of columns you want to query. If no column names are specified, all column names will be queried.

The output of the `ColAttributes()` method is a hash consisting of column names as keys and the attribute for the column as the key's value.

The code in [Example 7.18](#) will print out all the data types in a table called `foo` from the DSN called `MyDSN`. On line 14, the column's data types are retrieved with a call to the `ColAttributes()` method. The resulting hash is passed into a subroutine, `DumpAttrs()`, which prints out each column's data type.

### ***Example 7.18 Printing out a table's column data types***

```
01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "OZ";
03. my $Table = shift @ARGV || "Characters";
04. my $TableType = "'TABLE','VIEW','SYSTEM TABLE','".
05.           "'GLOBAL TEMPORARY','LOCAL TEMPORARY','".
06.           "'ALIAS','SYNONYM'";
07. my $db = new Win32::ODBC( $DSN );
08. if( ! $db )
09. {
10.   die "Error connecting: " . Win32::ODBC::Error() . "\n";
11. }
12. if( ! $db->Sql( "SELECT * FROM $Table" ) )
13. {
14.   if( $db->FetchRow() )
15.   {
16.     foreach my $Field ( $db->FieldNames() )
17.     {
18.       DumpAttrs( $db->ColAttributes( $db-
>SQL_COLUMN_TYPE_NAME, $Field ) );
19.     }
}
```

```

20.    }
21. else
22. {
23.     print "Fetch error: " . $db->Error();
24. }
25. }
26. else
27. {
28.     print "SQL Error: " . $db->Error();
29. }
30. $db->Close();
31.
32. sub DumpAttrs
33. {
34.     my( %Attributes ) = @_;
35.     foreach my $ColumnName ( sort( keys( %Attributes ) ) )
36.     {
37.         print "\t$ColumnName = $Attributes{$ColumnName}\n";
38.     }
39. }

```

## Data Source Names

Most administrators will create and manage a Data Source Name (DSN) by using the nifty GUI interface, such as the ODBC Administrator program or the Control Panel ODBC applet. Both of these (actually they are the same application) do a tremendous job of managing DSNs. Be aware, however, that at times you might need to programmatically manage DSNs. An administrator might want to write a Web-based CGI script enabling the management of DSNs, for example.

`Win32::ODBC` uses the `ConfigDSN()` function to do just this:

```
Win32::ODBC::ConfigDSN( $Action, $Driver, $Attribute1 [,
$Attribute2, ...] );
```

The first parameter (`$Action`) is the action specifier. The value of this parameter will determine what action will be taken. The valid actions and their values are as follows:

- `ODBC_ADD_DSN.` (0x01) Adds a new DSN
- `ODBC MODIFY_DSN.` (0x02) Modifies an existing DSN
- `ODBC REMOVE_DSN.` (0x03) Removes an existing DSN
- `ODBC_ADD_SYS_DSN.` (0x04) Adds a new system DSN
- `ODBC MODIFY_SYS_DSN.` (0x05) Modifies an existing System DSN
- `ODBC REMOVE_SYS_DSN.` (0x06) Removes an existing System DSN

In some versions of `Win32::ODBC`, the system DSN constants are not exported, so their values can be used instead.

The second parameter (`$Driver`) is the ODBC driver name that will be used. The driver must be one of the ODBC drivers installed on the computer. You can retrieve a list of available drivers by using either the `DataSources()` or the `Drivers()` function.

The remaining parameters are the list of attributes. These might differ from one ODBC driver to the next, and it is up to the programmer to know which attributes must be used for a particular DSN. Each attribute is constructed in the following format:

```
"AttributeName=AttributeValue"
```

These examples were taken from a DSN using the Microsoft Access ODBC driver:

```
"DSN=MyDSN"  
"UID=Cow"  
"PWD=Moo"  
"Description=My little bitty Data Source"
```

The `DSN` attribute is one that *all* ODBC drivers share. This attribute must be in the list of attributes you provide; otherwise, ODBC will not know what to call your DSN or, in the case of modifying and removing, which DSN you alter. It is wise to always include the `DSN` attribute as the first attribute in your list.

When you are adding or removing, you need only to specify the `DSN` attribute; others are not necessary. In the case of adding, any other attribute can be added later by modifying the DSN.

## Modifying DSNs

When you are modifying, you must include the `DSN` attribute so that ODBC will know which DSN you are modifying. Any additional attributes can either be added to the DSN or replace any attributes that already exist with the same name.

Some ODBC drivers require you to specify additional attributes (in addition to the `DSN` attribute) when using `ConfigDSN()`. When adding a new DSN that uses the Microsoft Access driver, for example, you must include the following database qualifier attribute:

```
"DBQ=C:\\SomeDir\\MyDatabase.mdb"
```

In [Example 7.19](#), the `ConfigDSN()` function is used three times. The first time (line 10), `ConfigDSN()` creates a new DSN. The second time (line 18) `ConfigDSN()` modifies the new DSN by adding the password (`PWD`) attribute and changing the user (`UID`) attribute. The third call to the `ConfigDSN()` function (line 22) removes the DSN that was just created. This code is obviously not very useful because it creates and then removes a DSN, but it shows how to use the `ConfigDSN()` function.

### **Example 7.19 Adding and modifying a DSN**

```
01. use Win32::ODBC;  
02. my $DSN = "My DSN Name";  
03. my $User = "administrator";  
04. my $Password = "adminpassword";  
05. my $Dir = "C:\\Database";  
06. my $DBase = "mydata.mdb";  
07. my $Driver = "Microsoft Access Driver (*.mdb)";
```

```

08. if( Win32::ODBC::ConfigDSN( ODBC_ADD_DSN,
09.                               $Driver,
10.                               "DSN=$DSN",
11.                               "Description=A Test DSN",
12.                               "DBQ=$Dir\\$DBase",
13.                               "DEFAULTDIR=$Dir",
14.                               "UID=" ) )
15. {
16.   Win32::ODBC::ConfigDSN( ODBC MODIFY_DSN,
17.                           $Driver,
18.                           "DSN=$DSN",
19.                           "UID=$User",
20.                           "PWD=$Password");
21.
22.   Win32::ODBC::ConfigDSN( ODBC REMOVE_DSN,
23.                           $Driver,
24.                           "DSN=$DSN" );
25. }

```

Line 9 assigns the `$Driver` variable with the name of the ODBC driver to be used. This value should come from a value obtained with a call to `Win32::ODBC::Drivers()`. The reason for this is because this value could change based on localization. A German version of ODBC, for example, would require line 9 to be as follows:

```
$Driver = "Microsoft Access Treiber (*.mdb)";
```

The value returned by the `Drivers()` function is obtained from the ODBC driver directly, so the value will be correct for the locale.

The `ConfigDSN()` function returns `true` if it is successful; otherwise, it returns `false`.

## Tip

*If you do not know what attributes to use in a call to `ConfigDSN()`, you can always cheat! You just use the ODBC Administrator program or the Control Panel's ODBC applet and create a temporary DSN.*

*After you have completed this, you need to run the Registry Editor (`regedit.exe` or `regedt32.exe`). If you created a system DSN, open this key:*

```
HKEY_LOCAL_MACHINE\Software\ODBC\Your_DSN_Name
```

*If you created a user DSN, open this key:*

```
HKEY_CURRENT_USER\Software\ODBC\Your_DSN_Name
```

*The value names under these keys are the attributes you specify in the `ConfigDSN()` function.*

After a DSN has been created, you might want to review how it is configured. This is done by using `GetDSN()`:

```
Win32::ODBC::GetDSN( $DSN ); $db->GetDSN();
```

`GetDSN()` is implemented as both a function and a method. When called as a function, you must pass in a parameter that is the name of a DSN whose configuration will be retrieved.

When used as a method, nothing is passed in to `GetDSN()`. The DSN that the ODBC connection object represents will be retrieved.

A hash is returned consisting of keys that are the DSN's attribute keyword. Each key's associated value is the DSN's attribute value. These key/value pairs are the same as those used in the `ConfigDSN()` function.

It is possible to retrieve a list of available DSNs by using the `DataSources()` function:

```
Win32::ODBC::DataSources();
```

`DataSources()` returns a hash consisting of data source names as keys and ODBC drivers as values. The ODBC drivers represented in the hash's values are in a descriptive format that is used as the second parameter in a call to `ConfigDSN()`.

[Example 7.20](#) illustrates how to retrieve the list of available DSNs and how to use `ConfigDSN()` to remove these DSNs.

### **Example 7.20 Removing all DSNs**

```
01. use Win32::ODBC;
02. if( my %DSNList = Win32::ODBC::DataSources() )
03. {
04.     print "Removing:\n";
05.     foreach my $Name ( keys( %DSNList ) )
06.     {
07.         print "$Name = '$DSNList{$Name}'\n";
08.         if( ! Win32::ODBC::ConfigDSN( ODBC_REMOVE_DSN,
09.                                         $DSNList{$Name},
10.                                         "DSN=$Name" ) )
11.     }
12.     # If we were unable to remove the
13.     # DSN maybe it is a system DSN...
14.     Win32::ODBC::ConfigDSN( ODBC_REMOVE_SYS_DSN,
15.                             $DSNList{$Name},
16.                             "DSN=$Name" );
17. }
18. }
19. }
```

Notice how [Example 7.20](#) uses the names and drivers that make up the hash returned by the `DataSources()` function. These values are used as the DSN name and driver in the calls to `ConfigDSN()`.

## Returning Available ODBC Drivers

`Drivers()` is yet another DSN-related function:

```
Win32::ODBC::Drivers();
```

This function returns a hash consisting of available ODBC drivers and any attributes related to the driver. Note that these attributes are not necessarily the same as the ones you provide in `ConfigDSN()`.

The returned hash consists of keys that represent the ODBC driver name (in descriptive format), and the key's associated value contains a list of ODBC driver attributes separated by semicolons (;), such as this:

```
"Attribute1=Value1;Attribute2=Value2;..."
```

These attributes are really not that useful for the common programmer but might be of use if you are programming ODBC drivers or if you need to make sure that a particular driver is configured correctly.

### Note

*The attributes returned by the `Drivers()` function (the ODBC driver's configuration attributes) are not the same type of attributes used in the `ConfigDSN()` attributes (an ODBC driver's DSN attributes).*

## Case Study: Changing the Database Paths for All MS Access DSNs

Jane's boss came running into her office and told her that the junior administrator did something really devastating to one of her web servers. Somehow, he has managed to corrupt the `D:` drive, the one where all the Access database files were kept.

Jane realized that she could not take the server down to reinstall a drive until the weekend. She also began kicking herself for not having already installed the RAID subsystem.

To correct the problem, she had someone restore the database files from a tape backup onto the server's `E:` drive. She figured that all she had to do was change the ODBC DSNs to point to their respective databases on the `E:` drive rather than the `D:` drive. No problem ... until she realized that there were hundreds of DSNs!

So Jane sat down and wrote the Perl script in [Example 7.21](#).

### ***Example 7.21 Changing the database paths for all MS Access DSNs***

```

01. use Win32::ODBC;
02. my $OldDrive = "d:";
03. my $NewDrive = "e:";
04. # We are looking for Access databases
05. my $Driver = GetDriver( ".mdb" ) || Error( "finding the ODBC
driver" );
06. my( %DSNList ) = Win32::ODBC::DataSources();
07. foreach my $DSN ( keys( %DSNList ) )
08. {
09.     next if( $DSNList{$DSN} ne $Driver );
10.    my( %Config ) = Win32::ODBC::GetDSN( $DSN );
11.    if( $Config{DBQ} =~ s/^$OldDrive/$NewDrive/i )
12.    {
13.        my $fResult = 0;
14.        print "Modifying $DSN...";
15.        $fResult = Win32::ODBC::ConfigDSN( ODBC_CONFIG_DSN,
16.                                         $Driver,
17.                                         "DSN=$DSN",
18.                                         "DBQ=$Config{DBQ}" );
19.        if( ! $fResult )
20.        {
21.            # If the previous attempt to modify the DSN
22.            # failed then try again but using a system DSN.
23.            $fResult = Win32::ODBC::ConfigDSN( ODBC_CONFIG_SYS_DSN,
24.                                         $Driver,
25.                                         "DSN=$DSN",
26.                                         "DBQ=$Config{DBQ}" );
27.        }
28.        print (( $fResult )? "Successful\n" : "Failed\n");
29.    }
30. }
31.
32. sub Error
33. {
34.     my( $Reason ) = @_;
35.     die "Error $Reason: " . Win32::ODBC::Error() . "\n";
36. }
37.
38. sub GetDriver
39. {
40.     my( $Extension ) = @_;
41.     $Extension =~ s/(.\$\$\])/\\\$1/gs;
42.     if( my %Sources = Win32::ODBC::Drivers() )
43.     {
44.         foreach my $Driver ( keys( %Sources ) )
45.         {
46.             if( $Sources{$Driver} =~ /FileExtns=[^;]*$Extension/i )
47.             {
48.                 return( $Driver );
49.             }
50.         }
51.     }

```

```
52. }
```

The script in [Example 7.21](#) first seeks the driver description for the ODBC driver that recognizes databases with an `.mdb` extension. This is performed by calling a subroutine `Getdriver()` on line 5. The subroutine makes a call to `Win32::ODBC::Drivers()` to get a list of all installed drivers and then tests them, looking for one that has a keyword `FileExtns` that matches the specified file extension (lines 42 through 51). Notice that line 41 prepends any period, backslash, and dollar sign with an escaping backslash. This is so that when the `$Extension` variable is used in the regular expression (line 46), the characters are not interpreted.

After the script has the driver description, it retrieves a list of available DSNs (line 6) and compares their drivers with the target one.

If the drivers match, the DSN's configuration is obtained (line 10), and the database file is compared to see whether the database is on the old `D:` drive. If so, the drive is changed to `E:`. The DSN is then modified to use the new path by first modifying it as a user DSN (lines 15 through 18); if that fails, it then tries it as a system DSN (lines 23 through 26).

By running this, Jane could quickly fix all the DSNs on her server in just a few minutes and with no errors. If she had manually altered the DSNs through a graphical ODBC administrator program, it would have taken much longer and would have been prone to mistakes. Jane also added a task to her calendar reminding her to install the server's RAID subsystem.

## Miscellaneous Functions

The ODBC API has a multitude of functions, many of which the `Win32::ODBC` extension exposes to a Perl script. A typical user will never use most of these, but for those who are migrating data or need to perform complex queries and cursor manipulation, among other tasks, these functions are required. This section discusses these functions and methods.

It is possible, for instance, to retrieve an array of column names using the `FieldNames()` method, although this is not the most useful feature because it only reports column names and nothing else. The syntax for the `FieldNames()` method is as follows:

```
@List = $db->FieldNames();
```

The returned array consists of the column names of the result set. There is no guarantee as to the order in which the names are listed.

## **Managing ODBC Connections**

Connections to data sources quite often have attributes that govern the nature of the connection. If a connection to a data source occurs over a network, for example, it might allow the network packet size to be changed. Likewise, logon timeout values, ODBC tracing, and transaction autocommit modes are considered to be connection attributes.

These attributes can be modified and examined by two `Win32::ODBC` methods: `GetConnectOption()` and `SetConnectOption()`:

```
$db->GetConnectOption( $Option );
$db->SetConnectOption( $Option, $Value );
```

For both methods, the first parameter (`$Option`) is the connection option as defined by the ODBC API. Appendix B contains a list of connection options.

The second parameter in `SetConnectOption()` (`$Value`) indicates the value to set for the specified option.

The return value for `GetConnectOption()` is the value for the specified option.

### **Warning: GetConnectOption()'s Return Value**

*Be careful with this because `GetConnectOption()` does not report any value to indicate an error—if the method fails, it will still return some value that might be invalid.*

The return value for `SetConnectOption()` is either `TRUE` if the option was successfully set or `FALSE` if the method failed to set the option.

In [Example 7.22](#), the ODBC tracing state is queried in line 5. (Tracing is where all ODBC API calls are logged into a text file so you can later see what your ODBC driver was doing.) If ODBC tracing is already active, the current trace file is retrieved (line 13) and is printed. Otherwise, the trace file is set (line 7), and tracing is turned on (line 8).

Enabling tracing will trace the ODBC driver until the process ends or until tracing is turned off. However, if tracing is turned on through the ODBC Control Panel applet, it enables tracing for all ODBC drivers. In this case, a script could disable the tracing for the script's process until the script finishes.

### **Example 7.22 Using the `GetConnectOptions()` and `SetConnectOptions()` methods**

```
01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "MyDSN";
03. my $db = new Win32::ODBC( $DSN ) || die "Error: " .
Win32::ODBC::Error();
04. my $TraceFile = "C:\\TEMP\\TRACE.SQL";
05. if($db->GetConnectOption( $db->SQL_OPT_TRACE ) == $db-
>SQL_OPT_TRACE_OFF )
06. {
07.   $db->SetConnectOption( $db->SQL_OPT_TRACEFILE, $TraceFile );
08.   $db->SetConnectOption( $db->SQL_OPT_TRACE, $db-
>SQL_OPT_TRACE_ON );
09.   print "ODBC tracing is now active.\n";
10. }
11. else
12. {
13.   $TraceFile = $db->GetConnectOption( $db->SQL_OPT_TRACEFILE );
14.   print "Tracing is already active.\n";
15. }
```

```
16. print "The ODBC tracefile is '$TraceFile'.\n";
17. # ...continue with your code...
18. $db->Close();
```

## Note

The ODBC API specifies many options that are quite useful (and, in some cases, necessary) for a user but that are far beyond the scope of this book. Appendix B lists most of these options and includes a brief description of them. Some of these descriptions are quite technical so that ODBC programmers can understand their impact.

For those who need more information on the ODBC options and their values or for those who are just curious, it is highly recommended that you consult a good book on the ODBC API. A couple of recommended books are Microsoft Press's *ODBC SDK and Programmer Reference* and Kyle Geiger's *Inside ODBC* (New Riders).

## Managing ODBC Statement Options

Just as an ODBC connection has attributes that can be queried and modified, so does an ODBC statement. When a script creates a `Win32::ODBC` object, it has both connection and statement handles. This means you not only can manage the connection attributes, you can also manage statement attributes such as the cursor type, the query timeout, and the maximum number of rows in a dataset from a `SELECT` query. These statement attributes are managed by the `GetStmtOption()` and `SetStmtOption()` methods:

```
$db->GetStmtOption( $Option );
$db->SetStmtOption( $Option, $Value );
```

The first parameter is the statement option as defined by the ODBC API. Appendix B contains a list of these statement options.

The second parameter in `SetStmtOption()` indicates the value to set for the specified option.

The return value for `GetStmtOption()` is the value for the specified option.

## Warning

*Be careful with this because `GetStmtOption()` does not report any value to indicate an error. If the method fails, it will still return some value that might be invalid.*

The return value for `SetStmtOption()` is either `TRUE` if the option was successfully set or `FALSE` if the method failed to set the option.

In [Example 7.23](#), the `SQL_ROWSET_SIZE` statement option is set to 100. This will retrieve a rowset of no more than 100 rows every time `FetchRow()` is called. Because the rowset size is greater than 1,

the actual row processed after `FetchRow()` will increase by 100. This is why we are using `GetStmtOption()` with the `SQL_ROW_NUMBER` option to determine the current row number.

### **Example 7.23 Using the `GetStmtOption()` and `SetStmtOption()` methods**

```
01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "MyDSN";
03. my $Table = shift @ARGV || "MyTable";
04. my $db = new Win32::ODBC( $DSN ) || die "Error: " .
Win32::ODBC::Error();
05. if( ! $db->Sql( "SELECT * FROM $Table" ) )
06. {
07.     $db->SetStmtOption( $db->SQL_ROWSET_SIZE, 100 );
08.     while( $db->FetchRow( 1, SQL_FETCH_NEXT ) )
09.     {
10.         my( %Data ) = $db->DataHash();
11.         # ...process data...
12.         if( my $Row = $db->GetStmtOption( $db->SQL_ROW_NUMBER ) )
13.         {
14.             print "Processed row $Row.\n";
15.         }
16.         else
17.         {
18.             print "Unable to determine row number.\n";
19.         }
20.     }
21. }
22. $db->Close();
```

### **Obtaining General ODBC Information**

There is one additional method that will retrieve information pertaining to the ODBC driver: `GetInfo()`. The information retrieved by `GetInfo()` is readonly and cannot be set.

```
$db->GetInfo( $Option );
```

The only parameter (`$Option`) is a value that represents the particular information desired. Appendix B provides a list of values. [Example 7.24](#) shows how `GetInfo()` can be used to determine whether the data source is read-only.

### **Example 7.24 Using the `GetInfo()` method**

```
01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "MyDSN";
03. my $db = new Win32::ODBC( $DSN ) || die "Error: " .
Win32::ODBC::Error();
04. if( "Y" eq uc $$db->GetInfo( $db->SQL_DATA_SOURCE_READ_ONLY ) )
05. {
06.     print "OOOPS! This data source is read only!\n";
07. }
08. else
```

```

09. {
10.   print "The datasource is read/write.\n";
11. }
12. $db->Close();

```

## **Checking for ODBC Function Support**

Not all ODBC drivers support all ODBC functions. This can cause problems for script writers. To check whether a connection supports a particular ODBC function, you can use the `GetFunctions()` method:

```
%Functions = $db->GetFunctions( [$Function1[, $Function2, ...]]);
```

The optional parameters are constants that represent ODBC functions such as `SQL_API_SQLTRANSACT` (which represents the ODBC API function `SQLTransact()`). There can be any number of functions passed in as parameters. If no parameters are passed in, all functions are checked.

A hash is returned consisting of keys that represent an ODBC API function, and the key's value is either `TRUE` or `FALSE`. If parameters were passed into the method, the resulting hash consists of only the keys that represent the parameters passed in.

In [Example 7.25](#), the `GetFunctions()` method is used to learn whether the ODBC driver supports transaction handling by means of the ODBC API's `SQLTransactions()` function. If it does, the Perl script can call `$db->Transaction()`.

### **Example 7.25 Using the `GetFunctions()` method**

```

01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "MyDSN";
03. my $db = new Win32::ODBC( $DSN ) || die "Error: " .
Win32::ODBC::Error();
04. my %Functions = $db->GetFunctions();
05. if( $Functions{$db->SQL_API_SQLTRANSACT} )
06. {
07.   print "Hey, this ODBC driver supports the SQLTransact()
function!\n";
08. }
09. else
10. {
11.   print "This ODBC driver does NOT support the SQLTransact()
function.\n";
12. }
13. $db->Close();

```

## **Limits on Column Size**

When a query returns a result set, a buffer must be created for each column. Generally, `Win32::ODBC` can determine the size of the buffer based on the column data type. Some data types,

however, do not describe the size of their data (such as the memo data type found in MS Access). In these cases, `Win32::ODBC` allocates a buffer of a predetermined size. This size is the maximum size that a buffer can be. This limit, however, can be both queried and changed with the `GetMaxBufSize()` and `SetMaxBufSize()` methods:

```
$db->GetMaxBufSize();  
$db->SetMaxBufSize( $Size );
```

The `SetMaxBufSize()` method takes one parameter (`$Size`), which represents the size in bytes that the limit of a buffer can be. Both functions return the number of bytes to which the current buffer size is limited.

### ***Retrieving Data Type Information***

Because each database has its own way of handling data, it can become quite difficult to know how to manage a particular data type. One database might require all text literals to be enclosed by single quotation marks, whereas another database might require double quotation marks. Yet the `MONEY` data type might require a prefix of some character such as the dollar sign (`$`) but nothing to terminate the literal value. Because a script using ODBC must be able to interact with any kind of database, it is important to be able to query the database to learn this information. This is where `GetTypeInfo()` comes in:

```
$db->GetTypeInfo( $DataType );
```

The first, and only, parameter is a data type. This value can be any one of the following data types:

<code>SQL_ALL_TYPES</code>	<code>SQL_TYPE_DATE</code>
<code>SQL_CHAR</code>	<code>SQL_TYPE_TIME</code>
<code>SQL_VARCHAR</code>	<code>SQL_TYPE_TIMESTAMP</code>
<code>SQL_LONGVARCHAR</code>	<code>SQL_INTERVAL_MONTH</code>
<code>SQL_DECIMAL</code>	<code>SQL_INTERVAL_YEAR</code>
<code>SQL_NUMERIC</code>	<code>SQL_INTERVAL_YEAR_TO_MONTH</code>
<code>SQL_SMALLINT</code>	<code>SQL_INTERVAL_DAY</code>
<code>SQL_INTEGER</code>	<code>SQL_INTERVAL_HOUR</code>
<code>SQL_REAL</code>	<code>SQL_INTERVAL_MINUTE</code>
<code>SQL_FLOAT</code>	<code>SQL_INTERVAL_SECOND</code>
<code>SQL_DOUBLE</code>	<code>SQL_INTERVAL_DAY_TO_HOUR</code>
<code>SQL_BIT</code>	<code>SQL_INTERVAL_DAY_TO_MINUTE</code>
<code>SQL_BIGINT</code>	<code>SQL_INTERVAL_DAY_TO_SECOND</code>
<code>SQL_BINARY</code>	<code>SQL_INTERVAL_HOUR_TO_MINUTE</code>
<code>SQL_VARBINARY</code>	<code>SQL_INTERVAL_HOUR_TO_SECOND</code>
<code>SQL_LONGVARBINARY</code>	<code>SQL_INTERVAL_MINUTE_TO_SECOND</code>

If successful, the `GetTypeInfo()` method returns `true`, and a dataset that describes the data type passed is returned; otherwise, the method returns `false`.

Any resulting dataset will contain one row representing the specified data type. Use the `FetchRow()` and `DataHash()` methods to walk through the resulting dataset. [Table 7.4](#) describes the columns of

the dataset. If `SQL_ALL_TYPES` is specified as the data type, the dataset will contain a row for every data type that the data source is aware of.

## GetTypeInfo() Availability

The `GetTypeInfo()` method first appeared in version 970208. Perl's `libwin32` standard library (as of version 0.16) does not seem to support this method.

Newer versions of the extension (available from <http://www.roth.net/perl/odbc/>) expose this method. If your version does not, you can manually add support for it. Refer to the note in the SQL "Delimiters" section earlier in this chapter.

[Example 7.26](#) illustrates the use of the `GetTypeInfo()` method. Line 5 assigns the value `SQL_ALL_TYPES` to the variable `$Type`. This could be any valid data type constant from the preceding list. Notice the hack to obtain that value, referring to it as a method from the `$db`. This is necessary because the data type constants are not exported from the `ODBC.PM` file (unless you edit `ODBC.PM` and add the constants to the `EXPORT` list).

**Table 7.4. Dataset Returned by the `GetTypeInfo()` Method**

Column Name	Description
<code>TYPE_NAME</code>	Data source-dependent name (such as MONEY). This is a text value.
<code>DATA_TYPE</code>	The SQL equivalent data type. This is an integer value.
<code>COLUMN_SIZE</code>	The maximum column size (in bytes) that the data source supports for the data type.
<code>LITERAL_PREFIX</code>	The string used to prefix literal value. On most data sources, for example, a char data type would specify a single quotation mark (').
<code>LITERAL_SUFFIX</code>	The string used to terminate a literal value. On most data sources, for example, a char data type would specify a single quotation mark (').
<code>CREATE_PARAMS</code>	A comma-separated list of keywords required when specifying this data type (as described by the <code>TYPE_NAME</code> column). The order of these keywords is the order to specify. The <code>DECIMAL</code> data type, for example, would return <code>"precision, scale"</code> . This indicates that when you specify a <code>DECIMAL</code> data type (as when creating a table), you must supply precision and scale values.
<code>NULLABLE</code>	Whether the data type can be <code>NULL</code> . Possible values are <code>SQL_NO_NULLS</code> , <code>SQL_NULLABLE</code> , <code>SQL_NULLABLE_UNKNOWN</code> .
<code>CASE_SENSITIVE</code>	Whether the data type is case sensitive. Possible values include <code>SQL_TRUE</code> and <code>SQL_FALSE</code> .
<code>SEARCHABLE</code>	This value illustrates how the data type is used in a <code>WHERE</code> clause.
<code>UNSIGNED_ATTRIBUTE</code>	Whether the data type is unsigned. Possible values are <code>SQL_TRUE</code> , <code>SQL_FALSE</code> , and <code>NULL</code> (if not applicable, such as for a char type).

**Table 7.4. Dataset Returned by the `GetTypeInfo()` Method**

Column Name	Description
<code>FIXED_PREC_SCALE</code>	Whether the data type has a predefined fixed precision and scale such as <code>MONEY</code> . Possible values are <code>SQL_TRUE</code> and <code>SQL_FALSE</code> .
<code>AUTO_UNIQUE_VALUE</code>	Whether the data type is auto-incrementing (such as a counter). Possible values include <code>SQL_TRUE</code> <code>SQL_FALSE</code> <code>NULL</code> (if this is not applicable or if the data type is a character).
<code>LOCAL_TYPE_NAME</code>	A localized (based on a language such as English or German) version of the data type's name.
<code>MINIMUM_SCALE</code>	The minimum scale of the data type or <code>NULL</code> if it's not applicable.
<code>MAXIMUM_SCALE</code>	The maximum scale of the data type or <code>NULL</code> if it's not applicable.

**Example 7.26 Determining how a SQL literal is handled using `GetTypeInfo()`**

```

01. use vars qw( %Data $Example );
02. use Win32::ODBC;
03. my $DSN = shift @ARGV || "MyDSN";
04. my $db = new Win32::ODBC( $DSN ) || die "Error: " .
Win32::ODBC::Error();
05. if( $db->GetTypeInfo( $db->SQL_ALL_TYPES ) )
06. {
07.     $~ = "DATA_HEADER";
08.     write;
09.     $~ = "DATA_ROW";
10.    while( $db->FetchRow() )
11.    {
12.        local %Data;
13.        local $Example;
14.        %Data = $db->DataHash();
15.        $Example = $Data{LITERAL_PREFIX} . "data" .
$Data{LITERAL_SUFFIX};
16.        write;
17.    }
18. }
19. else
20. {
21.     print "Can't retrieve type information: " . $db->Error() .
"\n";
22. }
23.
24. format DATA_HEADER =
25. Data Type          Prefix  Suffix   Example of data type
26. -----  -----  -----  -----
27. .
28.
29. format DATA_ROW =
30. @<<<<<<<<<<<< @<<<<<< @<<<<<< @<<<<<<<<<<<
```

```
31. $Data{TYPE_NAME}, $Data{LITERAL_PREFIX}, $Data{LITERAL_SUFFIX},  
$Example  
32. .
```

## **Processing More Results**

If supported by your ODBC driver, you can submit multiple queries in one call to `|sql()`. If the query is successful, you would fetch and process the data from the first SQL statement and then call the `MoreResults()` method and repeat the process of fetching and processing the data:

```
$db->MoreResults();
```

The return value is either `true`, indicating that another result set is pending, or `false`, indicating that no more result sets are available. [Example 7.27](#) demonstrates using `MoreResults()`.

### **Example 7.27 Processing multiple result sets with `MoreResults()`**

```
01. use Win32::ODBC;  
02. my $DSN = shift @ARGV || "MyDSN";  
03. my $Table = shift @ARGV || "MyTable";  
04. my $Table2 = shift @ARGV || "MyTable2";  
05. # Define ODBC 3.0 constants that may not be defined  
06. # in your version of Win32::ODBC  
07. my $SQL_BATCH_SUPPORT = 121;  
08. my $SQL_BS_SELECT_PROC = 4;  
09. my $db = new Win32::ODBC( $DSN ) || die "Error: " .  
Win32::ODBC::Error();  
10. my $Result = $db->GetInfo( $SQL_BATCH_SUPPORT );  
11. if( ! ( $Result & $SQL_BS_SELECT_PROC ) )  
12. {  
13.   print "The ODBC driver doesn't support multiple selects in a  
query.\n";  
14.   $db->Close();  
15.   exit;  
16. }  
17. if( ! $db->Sql( "SELECT * FROM $Table SELECT * FROM  
$Table2" ) )  
18. {  
19.   do  
20.   {  
21.     my $iCount = 0;  
22.     while( $db->FetchRow() )  
23.     {  
24.       my( %Data ) = $db->DataHash();  
25.       my $Key = (keys( %Data ))[0];  
26.       print ++$iCount, " ) $Key => '$Data{$Key}'\n";  
27.     }  
28.   } while( $db->MoreResults() );  
29. }  
30. else  
31. {
```

```

32.     print "Error in query: " . $db->Error() . '\n';
33. }
34. $db->Close();

```

## ODBC Transaction Processing

By default, most ODBC drivers are in *autocommit mode*. This means that when you perform an `INSERT`, `UPDATE`, `DELETE`, or some other query that modifies the database's data, the changes are made immediately; they are *automatically committed*. For some situations, however, this is unacceptable. Consider a CGI-based shopping cart script. Suppose that this script submits an order by adding information to a table in a database. Then it updates another table that describes the particular customer (the time he placed the order, what that order number was, and so forth). This is a case in which autocommit mode can be a problem. Suppose that, for some odd reason, the second update fails. The script will tell the user that the order failed, but the first update has already been submitted, so the order is now queued to be processed.

This situation can be avoided by turning autocommit mode off. When autocommit is off, you can submit as many modifications to the database as you want without having the modifications actually committed until you explicitly tell it to do so with the `Transact()` method:

```
$db->Transact( $Type )
```

The only parameter (`Type`) is the type of transaction. This can be either of the following listed values:

- `SQL_COMMIT`. All modifications are committed and saved to the database.
- `SQL_ROLLBACK`. All modifications are ignored, and the database remains as it was before any modifications were made to it.

The `Transact()` method will return `true` if the transaction was successful and `false` if it failed.

### Note

*Not all ODBC drivers support the `Transact()` method. You can use the `GetFunctions()` method to check whether your driver supports it.*

Before using the `Transact()` method, you need to make sure the autocommit mode is turned off. [Example 7.28](#) describes how to turn off autocommit as well as how to use `transact()`.

### Example 7.28 Using the `Transact()` method

```

01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "MyDSN";
03. my $db = new Win32::ODBC( $DSN ) || die "Error: " .
Win32::ODBC::Error();
04. $db->SetConnectOption( $db->SQL_AUTOCOMMIT, $db-
>SQL_AUTOCOMMIT_OFF );

```

```

05. if( ! $db->Sql( "INSERT INTO Foo (Name, Age) VALUES ('Joe',
31 )" ) )
06. {
07.     my $fSuccess = 0;
08.
09.     #...process something that sets $fSuccess...
10.
11.    if( 1 == $fSuccess )
12.    {
13.        $db->Transact( $db->SQL_COMMIT );
14.    }
15.    else
16.    {
17.        $db->Transact( $db->SQL_ROLLBACK );
18.    }
19. }
20. else
21. {
22.     print "Error with query: " . $db->Error() . "\n";
23. }
24. $db->Close();

```

## Row Counts

Some SQL statements (`UPDATE`, `INSERT`, `DELETE`, and sometimes `SELECT`) return a value that represents the number of rows that were affected by the statement. Not all drivers support this. If it is supported, however, you can obtain this number with the `RowCount()` method:

```
$db->RowCount();
```

The return value is either the number of rows affected or `-1` if the number of affected rows is not available. The `-1` value can be the result of an ODBC driver that does not support this function (such as the Access driver).

## Advanced Features of *Win32::ODBC*

So far, this chapter has discussed the simplest usage of the `Win32::ODBC` extension. This is how most scripts use it; however, `Win32::ODBC` provides access to most of the ODBC 2.0 API. This means that if you are familiar with ODBC, you can control your interactions with databases using some pretty powerful features, including fetching rowsets, cursor control, and error processing.

### Managing Cursors

If you are familiar with SQL, you'll be happy to know that cursors are supported. If you have no idea what cursors are, you probably don't need to be concerned about them. Cursors are beyond the scope of this book, so very little time will be spent on this topic. Several ODBC functions pertain to their use, however, and these are described in this section.

Basically, a *cursor* is an indicator that points to the current row in a given rowset. This is just like a cursor on a DOS window; it shows you where your current position in the DOS window is. The ODBC cursor just shows you the location of the current row from which you will be retrieving data.

`Win32::ODBC`, by default, resets the state of all cursors automatically for you whenever you use `Sql()` or any other method that returns rows of data (such as `GetTypeInfo()` and any of the cataloguing functions). Whenever these methods are used, the current cursor is dropped—that is, it is destroyed and forgotten.

This automatic dropping can be a problem when you need to keep a cursor open or if you have named a cursor and need to retain its name. This handling of the cursor can be overridden by changing the statement close type with the following method:

```
$db->SetStmtCloseType( $CloseType[, $Connection] );
```

The first parameter (`$CloseType`) is one of the close types documented in the following list:

- **`SQL_CLOSE`.** The current statement will not be destroyed, only the cursor. For all practical reasons, this is the same as `SQL_DROP`.
- **`SQL_DROP`.** The current statement is destroyed as well as the cursor.
- **`SQL_DONT_CLOSE`.** This will prevent the cursor from being destroyed any time new data is to be processed.
- **`SQL_UNBIND`.** All bound column buffers are unbound. This is of no use and is only included for completeness.
- **`SQL_RESET_PARAMS`.** All bound parameters are removed from their bindings. Because `Win32::ODBC` does not yet support parameter binding, this is of no use and is only included for completeness.

The optional second parameter (`$Connection`) is the connection number for an existing ODBC connection object. If this is empty, the current object is used. This is the object whose close type will be set.

The `SetStmtCloseType()` method will return a text string indicating which close type is set on the object.

Yet another method that will enable you to retrieve the current close type on a connection object is as follows:

```
$db->GetStmtCloseType( [$Connection] );
```

The optional first parameter (`$Connection`) is an ODBC connection object number that indicates which object will be queried. If nothing is passed in, the current object is assumed.

Just like the `SetStmtCloseType()` method, `GetStmtCloseType()` will return a text string indicating which of the five values documented in the list for `SetStmtCloseType()` is the close type.

A connection's cursor can be dropped by force if you need to. This can be very handy if you have previously set its close type to `SQL_DONT_CLOSE`:

```
$db->DropCursor( [$CloseType] );
```

The only optional parameter (`$CloseType`) indicates how the cursor is to be dropped. Valid values are the same as documented for `SetStmtCloseType()`, with the exception of `SQL_DONT_CLOSE`; this value is not allowed. If no value is specified, `SQL_DROP` is assumed.

Note that this will drop not only the cursor but also the current statement and any outstanding and pending result sets for the connection object.

All cursors created are given a name either by the programmer or by the ODBC driver. This name can be useful in queries and other SQL statements that allow you to use the cursor name such as `UPDATE table ... WHERE CURRENT OF cursor name`. To retrieve the cursor name, you use the `GetCursorName()` method:

```
$db->GetCursorName();
```

The method will return a text string that is the name of the cursor. If no cursor is defined, the method returns an `undef`.

If you want, you can name the cursor yourself. This might be necessary for stored procedures that require particular cursor names to be set:

```
$db->SetCursorName( $Name );
```

The first and only parameter (`$Name`) is the name of the cursor. Each ODBC driver defines the maximum length of its cursor names, but the ODBC API recommends not exceeding 18 characters in length.

If the cursor's name is successfully set, the method returns `TRUE`; otherwise, it returns `FALSE`.

## Note

*A cursor's name will be lost when the cursor is reset or dropped. For this reason, it is important that you set the statement close type to `SQL_DONT_CLOSE` before you set the cursor name. Otherwise, the moment the SQL query is generated, the cursor will be destroyed and the name will be lost. Win32::ODBC closes cursors before executing SQL statements unless the close type is set to `SQL_DONT_CLOSE`. You can always force the cursor to be dropped with the `DropCursor()` method.*

## Fetching a Rowset

Suppose you submit a query to a SQL Server. This query will return 10,000 rows, and you need to process each row. Every time you use the `FetchRow()` method, a command will be sent to the server requesting the next row. Depending on the network traffic, this can be quite a slow process because your ODBC driver must make 10,000 network requests and the SQL Server must respond with data 10,000 times. Add to this all the data required to package this network data up (such as TCP/IP headers and such), and you end up with quite a bit of network traffic. To top it all off, each

time you fetch the next row, your script must wait until the request has made it back from the server. All this can cause inefficient and slow response times.

If you could convince the ODBC driver to always collect 1,000 rows from the server each time a `FetchRow()` is called, the driver would only have to call to the server 10 times to request data.

This is where the concept of a rowset comes in. When you fetch rows, the ODBC driver really fetches what is known as a *rowset*. A rowset is just a collection of rows. By default, a rowset consists of one row. This configuration typically suffices, but it can be changed.

You can change the number of rows that make up a rowset by calling the `SetConnectOption()` method and specifying the `SQL_ROWSET_SIZE` option. When done, the advanced options of the `FetchRow()` method can be used to obtain a desired rowset. Then, by just resetting the rowset size to 1, a regular call to `FetchRow()` will retrieve one row at a time. See the next section, "[Advanced Row Fetching](#)," as well as [Example 7.19](#) for more details.

## Advanced Row Fetching

Although the `FetchRow()` method was described earlier, you need to revisit it now. A few parameters that can be passed into it need some explaining:

```
( $Result[, @RowResults] ) = $db->FetchRow( [ $Row [, $Type] ] );
```

If no parameters are passed in, the method will act as described earlier in the section "Processing the Results."

The first parameter (`$Row`) is a numeric value that indicates a row number.

The second value (`$Type`) indicates the mode in which the row number (the first parameter) will be used. The following list shows possible values for this parameter. If this parameter is not specified, the default value of `SQL_FETCH_RELATIVE` will be used.

- **`SQL_FETCH_FIRST`**. Fetch the first rowset in the dataset.
- **`SQL_FETCH_NEXT`**. Fetch the next rowset (typically, this is just one row) in the dataset. This will disregard the row value passed into `FetchRow()`. If this is the first `FetchRow()` to be called, this is equivalent to `SQL_FETCH_FIRST`.
- **`SQL_FETCH_LAST`**. Fetch the last rowset in the dataset.
- **`SQL_FETCH_PRIOR`**. Fetch the previous rowset in the dataset. If the cursor is positioned beyond the dataset's last row, this is equivalent to `SQL_FETCH_LAST`.
- **`SQL_FETCH_ABSOLUTE`**. Fetch the rowset that begins at the specified row number in the dataset. If the row number is 0, the cursor is positioned before the start of the result set (before physical row number 1).
- **`SQL_FETCH_RELATIVE`**. Fetch the rowset beginning with the specified row number in relation to the beginning of the current rowset. If row is 0, the current rowset will not change (instead, it will be updated).
- **`SQL_FETCH_BOOKMARK`**. This is not supported but is included for completeness.

The `FetchRow()` method will move the rowset to the new position based on the row and mode parameter.

There are two return values for the `FetchRow()` method. The first indicates whether the fetch was successful. It will return `TRUE` if successful; otherwise, it returns `FALSE`—unless, however, optional parameters are passed into the method. If this is the case, the first return value will be `SQL_ROW_SUCCESS` if successful; otherwise, it will be some other value.

The second return value is only returned if optional parameters are passed into the method. This second return value is an array of values. The array consists of a return value for each row fetched in the rowset. By default, ODBC drivers use a value of 1 for the rowset size so that `FetchRow()` will only return one value (because it only fetches one row). If you change the rowset to greater than one, however, your return value array will reflect the number of rows in your rowset.

Each value in the returned array corresponds with one of the values in the following list:

- `SQL_ROW_SUCCESS`. The row was successfully fetched.
- `SQL_ROW_UPDATED`. The row has been updated since the last time it was retrieved from the data source.
- `SQL_ROW_DELETED`. The row has been deleted since the last time it was retrieved from the data source.
- `SQL_ROW_ADDED`. The row has been added since the last time it was retrieved from the data source.
- `SQL_ROW_ERROR`. The row was unable to be retrieved.

Note that it is possible to set your cursor to a type that will reflect alterations made to the data. Suppose, for example, you are retrieving all the rows from a data source with `SELECT * FROM Foo`. Your script then begins to retrieve rows one at a time. While your script is fetching row 100, another program has deleted row 195 from the database. When you get to row 195, your ODBC driver will try to fetch it even though it has been deleted. This is because, when you execute your query, the driver receives a list of row numbers, which it will fetch. Because row number 195 was included in this list, the driver will try to fetch it when requested to, regardless of whether it has since been deleted. The driver will report the fact that it has been deleted by returning a value of `SQL_ROW_DELETED` in the return array.

For [Example 7.29](#), assume the rowset size was set to 10 so that every time you perform a `FetchRow()`, 10 rows are actually fetched. When you fetch the rowset starting at 190, `FetchRow()` will return a 10-element array that will consist of the values shown in [Figure 7.1](#).

**Figure 7.1. The returned row result array from `FetchRow()`, indicating that the fifth row has been deleted.**

```
01. $RowResults[0] == SQL_ROW_SUCCESS;
02. $RowResults[1] == SQL_ROW_SUCCESS;
03. $RowResults[2] == SQL_ROW_SUCCESS;
04. $RowResults[3] == SQL_ROW_SUCCESS;
05. $RowResults[4] == SQL_ROW_SUCCESS;
06. $RowResults[5] == SQL_ROW_DELETED;
07. $RowResults[6] == SQL_ROW_SUCCESS;
08. $RowResults[7] == SQL_ROW_SUCCESS;
09. $RowResults[8] == SQL_ROW_SUCCESS;
10. $RowResults[9] == SQL_ROW_SUCCESS;
```

[Example 7.29](#) demonstrates how you can use the extended features of `FetchRow()` to jump ahead several rows in a result set.

In this example, line 4 passes in a few connection options into the `new()` function. This will set the connection options before the connection is made to the database. In this case, we are instructing ODBC to use ODBC's builtin cursor library instead of relying on the ODBC driver's cursor abilities (which is the default behavior). Not all ODBC drivers' cursor capabilities are well implemented, and sometimes a script can see dramatic performance improvements when using the built-in library.

Line 6 instructs `Win32::ODBC` to keep the cursor open until it is explicitly closed. This was previously discussed in the "[Managing Cursors](#)" section of this chapter. Without this line, the cursor would be dropped before the call to `Sql()`. This would result in the cursor's settings being reset, which would defeat the purpose of setting any cursor options in the first place.

Next, line 8 changes the cursor type option to *static*. This enables the script to move both forward and backward in the dataset. However, the data is fixed (or static), so any updates to the database will not be reflected in the script's data set until the query is rerun.

Line 9 sets the rowset size to be 100. This means that every time ODBC fetches data from the database, it does it in 100-row blocks. Line 10 simply queries for the rowset size that line 9 set. This is not really necessary, but it is used here to illustrate how to query for the current rowset size.

Line 11 submits a query to the database, and line 14 begins a loop. The loop starts by fetching the next rowset. Because the rowset size was set to 100 rows, this will force ODBC to fetch the next 100 rows of data from the database.

Line 18 starts a do/while loop that will print 10 consecutive records. After that has finished, the script will fetch the next 100 records and repeat the process. This will effectively cause the script to read 10 records, jump ahead 100 records, read 10 more records, and keep repeating until there is no more data to read.

Line 17 resets the rowset size to 1. This will conserve network bandwidth while the do/while loop only needs to fetch 1 record at a time. Line 20 retrieves some data from the current row, and line 22 discovers the current row number. All this is printed, and then line 24 fetches the next row relative to the current rowset. After the loop is finished printing the 10 records, the rowset size is reset (line 25), and the script continues the while loop that starts on line 14.

### ***Example 7.29 Advanced and simple example of `FetchRow()`***

```
01. use Win32::ODBC;
02. my $DSN = shift @ARGV || "MyDSN";
03. my $Table = shift @ARGV || "MyTable";
04. my $db = new Win32::ODBC( $DSN, &Win32::ODBC::SQL_ODBC_CURSORS
=>
. &Win32::ODBC::SQL_CUR_USE_ODBC ) || die "Error: " .
Win32::ODBC::Error();
05. # Prevent the cursor from closing automatically
06. $db->SetStmtCloseType( SQL_DONT_CLOSE );
07. # Change our cursor type to static (assuming the driver
supports it)
```

```

08. $db->SetStmtOption( $db->SQL_CURSOR_TYPE, $db-
>SQL_CURSOR_STATIC );
09. $db->SetStmtOption( $db->SQL_ROWSET_SIZE, 100 );
10. my $RowSize = $db->GetStmtOption( $db->SQL_ROWSET_SIZE );
11. if( ! $db->Sql( "SELECT * FROM $Table" ) )
12. {
13.     # Fetching next block of $RowSize records...
14.     while( scalar $db->FetchRow( 1, SQL_FETCH_NEXT ) )
15.     {
16.         my $iCount = 10;
17.         $db->SetStmtOption( $db->SQL_ROWSET_SIZE, 1 );
18.         do
19.         {
20.             my( %Data )= $db->DataHash();
21.             my $Key = ( keys( %Data ) )[0];
22.             my $Row = $db->GetStmtOption( $db->SQL_ROW_NUMBER );
23.             print "\t$Row) $Key => '$Data{$Key}'\n";
24.         } while( $db->FetchRow(1, SQL_FETCH_RELATIVE ) && --
$iCount );
25.         $db->SetStmtOption( $db->SQL_ROWSET_SIZE, 100 );
26.         $RowSize = $db->GetStmtOption( $db->SQL_ROWSET_SIZE );
27.         print "\nCollect next group of $RowSize records...\n";
28.     }
29. }
30. else
31. {
32.     print $db->Error();
33. }
34. $db->Close();

```

The operation in [Example 7.29](#) can save you quite a bit of both time and network bandwidth (if the ODBC driver is talking to a network database server) because the first `FetchRow()` (line 9) will position the cursor to point at row 9,000 right away. The alternative would be to walk through the database one row at a time until it got to the 9,000th row.

At this point, the column's data will be retrieved and printed for the rest of the remaining rows by using the simple `FetchRow()` method.

## Cloning ODBC Connections

ODBC connection objects do not share anything with each other. For example, if you create two objects (`$db1` and `$db2`), even from the same database, the two objects cannot communicate with each other. If `$db1` has created a dataset with a named cursor, `$db2` cannot access `$db1`'s data.

There is a way, however, for `$db1` to talk with `$db2` using cloning. When you *clone* a `Win32::ODBC` object, you are creating a duplicate object. This cloned object talks with the same data source and, for all practical matters, is the same as the original object. Because these objects are separate and discrete from each other, they can run separate queries and process results as if they were two totally separate connections.

The wonderful nature of cloned objects is that they can talk with each other. If one object issues a query that produces a dataset, the other object can use that dataset for its own query. Cloned objects are created using the `new` command:

```
new Win32::ODBC( $Object );
```

If you pass in another `Win32::ODBC` connection object rather than a DSN name, the object will be cloned—that is, another object will be created that shares the same connection to the database. In technical terms, the objects will share the same environment and connection handles although their statement handles will be unique.

This is used mostly in conjunction with cursor operations.

## Error Processing

An ODBC error is an error generated by either the ODBC Manager or an ODBC driver. This is a number that refers to an error that occurred internally to ODBC. In practice, this error is really only useful if you have an ODBC API manual or information on error numbers for a particular ODBC driver. The SQL state, however, is a standardized string that describes a condition to which all ODBC drivers adhere.

### ***Retrieving Errors***

`Win32::ODBC` tracks ODBC errors in two ways. The first way is that the module itself will always track the last ODBC error that occurred. The second way is that a particular ODBC connection object will track its own errors. You retrieve the error information by calling `Error()` either as an object method or as a module function:

```
Win32::ODBC::Error();
$db->Error();
```

If used as a module function (the first example), it will report the last error that the ODBC module had generated regardless of which connection was responsible for the error. If used as an object's method (the second example), however, the last error that the particular object generated will be reported.

When retrieving error information with the `Error()` method (or function), the results could be in one of two formats (depending on the context of the assignment): an array or a text string in a scalar context.

## **Array Context**

If a call were made to `Error()` in an array context:

```
@Errors = $db->Error();
```

The returning array, `@Errors`, would consist of the following elements:

```
( $ErrorNumber, $Tagged Text, $Connection Number, $SQL State)
```

[Table 7.5](#) lists a description for each element.

**Table 7.5. Fields Returned By the `Error()` Function and Method**

Field	Description
<code>Error Number</code>	The actual error number as reported by ODBC (not too useful unless you have an ODBC API manual). This number might be specific to an ODBC driver, so it could have one meaning to an Oracle driver but another meaning to a Sybase driver. When an error occurs, there is always an ODBC-specific error number. This is different from the SQL state because it identifies an error that ODBC generated. This error is not an ISO standard; instead, it is an ODBC-specific error.
<code>Tagged Text</code>	The text description of the error. Either the ODBC Manager or the ODBC driver reports this. This text is in a tagged format—that is, it identifies both the vendor and the component that generated the error. For example, the error text <code>[Microsoft][ODBC Microsoft Access 2.0 Driver] Unable to open database file</code> identifies that the Microsoft ODBC Access driver reported the error <code>Unable to open database file</code> . The first tag is the vendor, and the second tag is the ODBC component that generated the error. Another example, <code>[Microsoft][ODBC Manager] The data could not be retrieved because the database server is offline</code> , illustrates that the Microsoft ODBC Manager reported the error that the server is unavailable.
<code>Connection Number</code>	The number of the ODBC connection that generated the error. If there is no connection number, the number is left as an <code>undef</code> . Each <code>Win32::ODBC</code> object has an associated unique connection number. You can discover an ODBC connection object's connection number by calling the <code>Connection()</code> method.
<code>SQLState</code>	The SQL state of the error. This value complies with the ISO SQL-92 standard; hence a programmer can rely on it to represent a particular error condition, regardless of which ODBC driver generated the error. It would be a good idea to invest in a book on ODBC so that the SQL state will make sense.

## Scalar Context

If a call were made to `Error()` in a scalar context:

```
$Error = $db->Error();
```

The result would resemble the following string:

```
"[Error Number] [Connection Number] [Tagged Text]"
```

[Table 7.5](#) lists descriptions for these elements.

## The SQL State

The current SQL state can be retrieved by using the `SQLState()` method:

```
$db->SQLState();
```

The value returned constitutes the last SQL state of the particular ODBC connection object. Because the SQL state represents the state of the connection and is not driver specific (as the error number is), it is the same for all ODBC drivers. Any good book on the ODBC API will list all possible SQL state codes.

## Summary

Interacting with databases is not very difficult, but having to interact with several potentially different databases can cause nightmares because of a lack of conformity across them.

The ODBC API is commonly found on Win32 machines, and it addresses this issue of database conformity. By using ODBC, a programmer can create huge scripts that will work with any database and that can be easily transported from machine to machine.

The `Win32::ODBC` extension has provided thousands of CGI, administrative, and other types of scripts with access to ODBC. This extension provides a relatively thin layer of abstraction from the ODBC API that is ideal for anyone who is prototyping an application in Perl that will be programmed in C or C++ to access the ODBC API. Any coder who is familiar with the ODBC API will feel at home using this extension.

If a programmer is not familiar with ODBC, the extension hides most of the tedious work required so that the coder can focus on manipulating data and not worry about how to retrieve it.

You can also access ODBC data sources in other ways, such as by using the Perl Database Interface (DBI), which has a basic ODBC driver. Considering that the DBI ODBC extension exists on both Win32 and UNIX platforms, it is ideal for cross-platform scripts but lacks some of the functionality that `Win32::ODBC` provides. Additionally, the `Win32::OLE` extension (refer to [Chapter 5](#), "Automation") provides access into Microsoft's ActiveX database objects (ADO) and other COM-based ODBC implementations.

# Chapter 8. Processes

Perl scripts are quite powerful and can do pretty much anything you need them to. At times, however, you'll need a separate application to process data or some other task for you—something that might be impractical or impossible for Perl to do itself. Perl supplies basic functions that will create such new processes. These functions were inherited from the UNIX world. They are quite powerful and helpful, but the Win32 platforms provide options for process creation that are not supported by the standard Perl process-creation function.

This chapter investigates what is involved in creating a new process that makes use of the Win32 way of doing things. This chapter also looks at how to manage a process's priority, suspend state, and destruction.

In addition, this chapter looks at the difference between Perl file handles and Win32 handles. This is important to understand because some of the process-creation functions have options that involve these handles.

The Win32 extensions used in this chapter include `Win32::AdminMisc` and `Win32::Process`.

## The STD Handles

Perl makes use of what are known as the standard file handles: `STDIN`, `STDOUT`, and `STDERR`. Perl opens them automatically when a script starts. At this time, for example, unless otherwise specified, all output from a `print` command is being automatically routed to `STDOUT`, or the standard output device. In such context, the following commands are the same:

```
print "Hello.\n";
print STDOUT "Hello.\n";
```

If you do not specify a file handle to print to, `STDOUT` will be used. `STDOUT` is a special file handle that, by default, sends data to the standard output device—typically, the screen. The `STDIN` (standard input device) defaults to the keyboard, and `STDERR` (standard error device) defaults to the screen, just like `STDOUT`.

Perl is very good at enabling you to redirect these standard file handles to other file handles, files, sockets, and pipes. The Win32 API uses a different type of handle, however—one that can be inherited.

## Handle Inheritance

The concept of handle inheritance warrants some explanation. When you open a file, bind to a socket, create a named pipe, connect to a remote Registry, or do anything in which you receive some sort of identifying object, your process has been given a handle. Simply stated, a *handle* is a number that represents something. When you open a file, for instance, the operating system returns to you a number. Anytime you want to read from or write to that stream, you must tell the operating system what that number is. In Perl, this number is hidden so that everything is a little easier for the programmer. From the debugger, you can dump the contents of your Perl file handle as demonstrated in [Figure 8.1](#). Notice that the last line shows that the file handle `DATA` has a `fileno` (or file number) of 3. This number represents the open file.

**Figure 8.1. Discovering a file's handle in the debugger.**

```
C:\Perl\test>perl -d test.pl
Stack dump during die enabled outside of evals.

Loading DB routines from perl5db.pl patch level 0.95
Emacs support available.
Enter h or 'h h' for help.
main::(test.pl:2): open( DATA, "> stuff.txt" );
DB<1> n
main::(test.pl:3): print DATA "Hello!\n";
DB<1> X DATA
FileHandle(DATA) => fileno(3)
```

Processes can inherit Win32 handles. Notice that I am specifying Win32 handles, not Perl file handles; this is very important. When you create a new process, you can tell the new process to inherit the handles you have acquired in the current process (any open sockets, files, pipes, and so forth). If the new process you create is a Perl script, however, don't expect to be able to access those opened sockets and files you inherited. This is because a Perl file handle differs from a Win32 handle. You see, to open a file, Perl creates a file handle by asking the C language's runtime library to open the file. The C library, in turn, asks the operating system to open it. The Win32 API will return a handle (just a number that the OS associates with the open file), and the C library will store the handle into some data structure it creates. A pointer to this structure will be returned to Perl, which puts this pointer into yet another structure. Perl will then associate a Perl file handle, one that the programmer specifies, with this structure. Because a new process does not have access to memory in other processes, any inherited memory pointers won't point to the original memory structures.

Okay, so this is a bit technical, but the point is that a Win32 handle is not the same as a Perl file handle. Although a process can inherit Win32 handles, it does not help a Perl script much if it opens a file in one process and attempts to access the open file in another process because only the Win32 file handle (and not the Perl file handle) was inherited. The next section addresses this problem.

## Retrieving Win32 Standard Handles

When managing Win32 processes, occasions arise when a Win32 file handle needs to be specified. Because this file handle is not the same as a Perl file handle, you need a way to obtain a Win32 file handle.

The `Win32::AdminMisc` extension provides the capability to discover the Win32 file handle for all three of the standard file handles opened by default. You can retrieve the Win32 standard handles by utilizing the `Win32::AdminMisc::GetStdHandle()` function:

```
$Handle = Win32::AdminMisc::GetStdHandle( $HandleName );
```

The first parameter (`$HandleName`) can be one of the following three constants:

- `STD_INPUT_HANDLE`. The standard input handle (`STDIN`).
- `STD_OUTPUT_HANDLE`. The standard output handle (`STDOUT`).
- `STD_ERROR_HANDLE`. The standard error handle (`STDERR`).

If the `GetStdHandle()` function is successful, it returns a number that represents the Win32 handle. This value can be important for functions such as `Win32::AdminMisc::CreateProcessAsUser()`. If the `GetStdHandle()` function is not successful, it returns a `FALSE` value.

## Process Management

The Microsoft Visual C++ documentation defines a *process* as "an executing instance of an application. For example, when you double-click the Notepad icon, you start a process that runs Notepad." Processes have long been a necessity to Perl programmers, and the language reflects this by providing several commands that enable you to create processes:

- **System command.** `system( "program.exe" );`
- **Open command using piped output.** `open( PROGRAM , "program.exe | " );`
- **Open command using piped input.** `open( PROGRAM, " | program.exe" );`
- **Backtricks.** `@Output = `program.exe`;`

Each of these commands enables you to launch a program. The problem with these, however, is that the new process cannot be controlled. If you run the code in [Example 8.1](#), for instance, the Notepad program will start, but the Perl script will not complete until you quit the Notepad program. The script will wait until the Notepad process terminates. This temporary pause means your script will not be able to continue running while the new Notepad process runs. This can be more than just an inconvenience.

### **Example 8.1 Creating a new process using the `system()` command**

```
01. my $Program = "notepad.exe";
02. print "Running $Program...\n";
03. system( $Program );
04. print "Finished.\n";
```

This lack of simultaneity can be corrected by using the `start` command built into the Windows NT and Windows 95 command processor. Using the `start` command creates the new process independently from the Perl script that spawned it. This means that the script can continue running and even end regardless of the state of the newly created process. Running the code in [Example 8.2](#) will print "Running notepad.exe..." followed by "Finished." and then terminate. The Notepad program will start up in the meantime. You will notice that the Perl script is not tied in any way to the Notepad program, so when one ends, it does not affect the other.

### **Example 8.2 Starting an independently running process with the `system()` function**

```
01. my $Program = "start notepad.exe";
02. print "Running $Program...\n";
03. system( $Program );
04. print "Finished.\n";
```

## **Creating a Process with `Win32::Spawn()`**

Using the `start` command is the equivalent of using the `Win32::Spawn()` function. You can use either, but the `Spawn()` function will be just a little bit more efficient because it does not have to shell out to a command processor just to run each time. In addition to efficiency, the `Win32::Spawn()` function will return the process ID (PID) of the newly created process. This can be important if you want to later manage the new process. The syntax for the `Win32::Spawn()` function is as follows:

```
Win32::Spawn( $Program, $CommandLine, $PID );
```

The first parameter (`$Program`) is the path to the executable file that you are going to run. This can be a relative path, a full path, or a UNC. The full name of the executable, however, must be provided; you must use `notepad.exe` instead of simply using `notepad`.

The second parameter (`$CommandLine`) is the command-line version of the program. This is a bit odd, but it must be a string that contains whatever you would enter on a command line to execute the process, including both the program and any switches or command-line options. If you wanted to load a `readme.txt` file with Notepad, you would specify the path to `notepad.exe` as the first parameter and the second parameter would be `notepad readme.txt`. The first part of this string does not have to be the program you run (for technical reasons beyond the scope of this book), but it should make sense to the new process. You could use `MyFootHurts readme.txt` as the second parameter, and it would work just as well.

The third parameter (`$PID`) is set to the process ID of the process.

If the process is successfully created, a `1` will be returned; otherwise, a `0` is returned.

[Example 8.3](#) illustrates proper implementation of the `Win32::Spawn()` method.

### **Example 8.3 Creating a new process with `Win32::Spawn()`**

```
01. use Win32;
02. $App = "notepad.exe";
03. $Cmd = "notepad readme.txt";
04. if( Win32::Spawn( $App, $Cmd, $Pid ) )
05. {
06.     print "$Cmd was created with the process ID $Pid.\n";
07. }
08. else
09. {
10.     print "Could not start \"$Cmd\"\n";
11.     print "Error: " .
Win32::FormatMessage( Win32::GetLastError() );
12. }
```

"This is all fine and good," you might be thinking, "but what if I need to control the new process?" Well, that is where Perl falls short. In the UNIX world, you could run `ps` to find the process ID of a process and then call a `kill` to terminate it. Likewise, you could run the `nice` utility to change the priority under which the process runs, but this does not help us Win32 users. Sure, similar utilities are available, such as `tlist.exe` and `kill.exe`, but you cannot guarantee (unlike our UNIX counterparts) that each machine on which you run your script will have these nonstandard utilities. This is where the `Win32::Process` extension comes in.

With the `Win32::Process` extension, you create a blessed `Win32::Process` object that contains several methods, as described in the sections that follow.

#### **Note**

*In Perl, a blessed object is a variable that has been "blessed" by a particular module. This means the variable is a representative of the module and has its own set of member variables and methods (functions).*

When you create a new `Win32::Process` object, you are actually creating a new process that is a running program, as if you called the `system()` function. The real difference is that, with the object, you can manage the new process.

## Creating a Controllable Process

Suppose you wanted to create a new Notepad process. You could use the `Win32::Process::Create()` function:

```
Win32::Process::Create( $Process, $Program, $CommandLine, $Inherit,
$Flags, $Directory );
```

The first parameter (`$Process`) holds the process object that is created if the creation is successful. Note that whatever this parameter contains upon entry into the function is ignored.

The second parameter (`$Program`) is the path to the executable file you are going to run. This is the same as the first parameter to the `Win32::Spawn()` function.

The third parameter (`$CommandLine`) performs the same role as the second parameter of the `Win32::Spawn()` function.

The fourth parameter (`$Inherit`) is a Boolean value that specifies whether to inherit handles from the currently running process. A `1` means to inherit handles, and a `0` means to not inherit handles. This makes more sense if you read the section "Handle Inheritance."

The fifth and most exciting parameter (`$Flags`) describes how the new process should be created and how fast it should run. This parameter is created by logically OR'ing the constants found in [Table 8.1](#) and *only one* priority constant from [Table 8.2](#). The default creation flag action (if no constants from [Table 8.1](#) are specified) will be to use the same console as the current process for input and output. The default priority class is normal.

Finally, the sixth parameter (`$Directory`) specifies which directory the new process will use as its default directory. If the function is successful, it returns a true value (nonzero); otherwise, it returns false (zero).

**Table 8.1. Process Creation Flag Constants**

Constant	Description
<code>DEBUG_PROCESS</code>	This tells the new process (and any child processes it creates) that the process creating the new process, known as the debugger process, is debugging it. Anytime certain events take place in the new process or any of its children, known as debuggees, the debugger process will be notified. This debugging has to do with Win32 programs and is not in any way related to the Perl debugger. Most Perl coders will never have a need to use this flag.
<code>DEBUG_ONLY_THIS_PROCESS</code>	Same as <code>DEBUG_PROCESS</code> except that only the new process is

**Table 8.1. Process Creation Flag Constants**

Constant	Description
<code>CREATE_SUSPENDED</code>	in the debug state. Child processes of the new process are not in the debug state. Like the <code>DEBUG_PROCESS</code> flag, most Perl coders will never have a need to use this flag.
<code>DETACHED_PROCESS</code>	it is told to do so with the <code>Resume()</code> method. This is very handy for synchronizing events. See the section "Suspending a Process" for more details.
<code>CREATE_NEW_CONSOLE</code>	If the process being created is a console application (that is, a program that uses a DOS-like window for output), it will be created without any console window. You would want to use this if the application does not need either input or output.
<code>CREATE_NEW_PROCESS_GROUP</code>	If the process being created is a console application, a new console window will be created. This new window will be used for input and output.
	Use this flag to create a new Perl process with its own console window.
<code>CREATE_DEFAULT_ERROR_MODE</code>	The new process will start a new process group. All processes in a process group will receive notification of certain events, such as a Ctrl+C or Ctrl+Break. Now, before you get excited about this, you must know that to take advantage of this option, you must access the Win32 API
	<code>GenerateConsoleCtrlEvent()</code> function. You can find this in the <code>Win32::Console</code> and <code>Win32::API</code> extensions.
<code>CREATE_SEPARATE_WOW_VDM</code>	The new process is to use the default error mode. A discussion of a process's error mode is beyond the scope of this book, but if you are aware of what it is and how to use it, this flag will cause the new process to not inherit the current process's error mode.
<code>CREATE_UNICODE_ENVIRONMENT</code>	If the new process is a 16-bit Windows application (a program written for Windows 3.x), a separate virtual DOS machine will be used.
	Chances are, you will never have a need to use this. If you did use it, however, the new process would use Unicode characters for its environment variables.

**Table 8.2. Process Priority Constants**

Priority Class Constant	Description
<code>IDLE_PRIORITY_CLASS</code>	The process is given CPU time only when the system is idle—that is, when nothing else is really going on. Even when you have several

**Table 8.2. Process Priority Constants**

Priority Class Constant	Description
	applications running, there is still idle CPU time.
NORMAL_PRIORITY_CLASS	This is the class that a process becomes by default if no other priority is specified in the creation flags with a call to <code>Win32::Process::Create()</code> . When you run an application by double-clicking its icon, this is the class it runs under.
HIGH_PRIORITY_CLASS	A process with this priority class is much more responsive than the idle or normal classes. Any process running under this priority causes the other processes to noticeably slow down. Use this sparingly.
REALTIME_PRIORITY_CLASS	This is a special class that takes up so much CPU attention that it can cause some problems, such as preventing other processes from finishing I/O in a timely manner (hence the potential for data loss). This class was designed for tasks that are very short in duration and that need to not be interrupted during their execution. A user needs to have a special privilege (Increase Scheduling Priority) to set this priority class; typically, administrators have this. Kernel-level drivers (for example, the mouse and keyboard drivers) usually take advantage of this class.

You can see that using the `Create()` function is much more complicated than the `Win32::Spawn()` function, but it provides much more control. [Example 8.4](#) demonstrates this fact.

#### **Example 8.4 Using the Win32::Process::Create() function**

```

01. use Win32::Process;
02. my $App = "$ENV{windir}\notepad.exe";
03. my $Cmd = "$App c:\temp\readme.txt";
04. my $fInherit = 1;
05. my $Flags = 0;
06. my $Dir = ".";
07. my $fResult = Win32::Process::Create( $Process,
08.                                         $App,
09.                                         $Cmd,
10.                                         $fInherit,
11.                                         $Flags,
12.                                         $Dir);
13. if($fResult )
14. {
15.     my $Pid = $Process->GetProcessID();
16.     print "$Cmd has been created with a process ID of $Pid.\n";
17. }
18. else
19. {
20.     print "Unable to start $Cmd.\n";
21.     print "Error: " .
Win32::FormatMessage( Win32::GetLastError() );

```

```
22. }
```

## Note

*For those of you who are C programmers, it is interesting to note that the Win32::Process::Create() function creates a process by passing in NULL pointers for the process and thread security attributes. This means the process will be created with what is known as a default security descriptor, which means that anyone can access, kill, and change the priority of the process—that is, there is no security placed on the process.*

## The Process ID

After a process has been successfully created using `Win32::Process::Create()`, you can discover the process identifier (PID). Use the `GetProcessID()` method to do this:

```
$Process->GetProcessID();
```

This will return the PID that represents the process.

## Killing the Process

If there is a need to terminate a process you have created with `Win32::Process::Create()`, you can use the `Kill()` method:

```
$Process->Kill( $ExitCode );
```

This assumes that you have created a `Win32::Process` object called `$Process`.

The only parameter (`$ExitCode`) indicates what exit value is reported. Any process that queries the exit code will be provided this value. Typically, this value is used by a calling process such as a batch file.

The `Kill()` method makes use of the Win32 API's `TerminateProcess()` function, which has a reputation of being pretty rough. When you force a termination this way, the process is terminated, but any DLLs that have been loaded are not notified before they shut down. This prevents any last minute cleanup such as writing data to files or removing any temporary files that might have been created. Generally speaking, you don't want to use this unless you have to (see [Example 8.5](#)).

## Example 8.5 Killing a process

```
01. use Win32::Process;
02. my $App = "$ENV{windir}\notepad.exe";
03. my $Cmd = "$App c:\\temp\\myfile.txt";
04. my $bInherit = 1;
05. my $Flags = 0;
06. my $Dir = "c:\\perl\\test";
07. my $fResult = Win32::Process::Create( $Process,
```

```

08.         $App,
09.         $Cmd,
10.         $bInherit,
11.         $Flags,
12.         $Dir);
13.     if( $fResult )
14.     {
15.         print "$Cmd has been created.\n";
16.
17.         print "It's process ID is " , $Process->GetProcessID() , "\n";
18.
19.         sleep(12);
20.
21.         $Process->Kill( 0 );
22.
23.     }
24.
25.     print "The process $Cmd was not created! \n";
26.     print "Error: " .
Win32::FormatMessage( Win32::GetLastError() );
27. }
```

## Tip

*Now that you know how to kill a `Win32::Process` process you have created, it is good to know that you are not limited to killing only processes you created. You can call the `kill()` method as a function without specifying a process object. All you need to do is pass in a process's PID:*

```
Win32::Process::Kill( $MyProcessPid, $ExitCode );
```

*The process PID can be found a number of ways, such as manually looking at the Task Manager. Alternatively, you can run a process listing utility such as `ps.exe`. However, you would need to capture the output and parse it for the PID.*

*In the book `Win32 Perl Scripting: The Administrator's Handbook` (also published by New Riders), I cover several techniques you can use to programmatically discover what processes are running and how to procure their PIDs.*

## Suspending a Process

When you have created a process with `Win32::Process::Create()`, you might want to temporarily stop the process or pause it. This is handy when you need to synchronize processes or if the process is some daemon like a Web server that needs to be temporarily paused. Suspending a process is performed with the `Suspend()` method:

```
$PrevTotal = $Process->Suspend();
```

What really happens when you use `Suspend()` is that the process's primary thread is put into a suspended state—that is, the thread is not given any CPU time. This results in the process just sitting there and not doing anything.

### Note

*A thread is a dedication of CPU time to run a process. If you have multiple threads, you are requesting that the CPU not only multitask your process with other processes, but also multitask functions within your process. Having multiple threads in a process is similar to having multiple processes on an operating system. All Win32 applications have at least one thread, the so-called primary thread. This thread is the first (and for many applications, the only) thread that is created.*

A process can be suspended multiple times. Each time it is suspended, an internal suspension counter is incremented by one. This can occur up to a limit that the Win32 kernel defines, which happens to be 127. If you try to suspend a process more than 127 times, an error will occur. The return value of the `Suspend()` method is the total number of times the process has been suspended before the method was called. This means that when you use `Suspend()`, it might return a 0, but this does not mean an error; it just means that the process was suspended 0 times *before* the method was called. If you create a process with the `CREATE_SUSPENDED` flag, the process will be created and put into a suspended state automatically. This, of course, will increase the internal suspension counter to 1.

Because the suspension is performed by placing the primary thread of a process into a suspended state, it is possible that the process started other threads before you called the `Suspend()` method. (This would not be the case if you created the process with the `CREATE_SUSPENDED` flag.) Therefore, if you try to suspend some application, such as Microsoft Word, that uses multiple threads, don't be surprised if the application continues to run (or at least parts of it continue to run).

If the `Suspend()` method fails, it returns a -1 to indicate an error.

Example 8.6 in the next section demonstrates the use of the `Suspend()` method. In this case, a Web server process is created. Lines 18 through 28 cause the script to sleep for one minute and then check the number of network connections made to (and from) the machine. If the number is greater than 100 (actually 96 because `NETSTAT.EXE` outputs 4 lines of nonconnection information), the Web server is suspended for two minutes—long enough for the connections to time out.

This example is not very practical, but it does illustrate how the `Suspend()` and `Resume()` methods work.

## **Resuming a Suspended Process**

When a process has been suspended—that is, the internal suspension counter is greater than 0—the only way to take it out of its suspended state is to use the `Resume()` method:

```
$PrevTotal = $Process->Resume();
```

By calling this method, you are going to decrement the internal suspension counter for the primary thread of your process. When this internal counter becomes `0`, your process will continue to run where it left off. If you have called the `Suspend()` method five times, you must call the `Resume()` method five times before the process resumes execution. (The exception is if you created the process with the `CREATE_SUSPENDED` flag, which means you would have to call `Resume()` six times.)

`Resume()` will return the previous number of suspensions before the method was invoked. Therefore, if it returns a `1`, you know the process is no longer suspended. If there is an error, it will return `-1`.

It is possible for another application to suspend your process without your knowledge. Therefore, calling `Resume()` might not cause the process to leave the suspended state. If your script needs to indeed resume a suspended process, it is best to call `Resume()` until the return value is `1`. This indicates that the previous suspend call to the process was `1`, so your `Resume()` call decremented the suspension state to `0`. Line 26 of [Example 8.6](#) illustrates calling `Resume()` until it returns a value that is not greater than `1`.

### ***Example 8.6 Suspending and resuming a process***

```
01. use Win32::Process;
02. my $App = "c:\\webserver\\webserver.exe";
03. my $Cmd = "webserver 192.168.1.10";
04. my $fInherit = 1;
05. my $Flags = 0;
06. my $Dir = ".";
07. my $fResult = Win32::Process::Create( $Process,
08.                                         $App,
09.                                         $Cmd,
10.                                         $fInherit,
11.                                         $Flags,
12.                                         $Dir);
13. if( $fResult )
14. {
15.     print "$Cmd has been created.\n";
16.     print "It's process ID is ", $Process->GetProcessID(), "\n";
17.     my $Continue = 100;
18.     while( $Continue- )
19.     {
```

```

20.      sleep( 60 );
21.
22.      my @Output = `netstat`;
23.      if( scalar @Output > 100 )
24.      {
25.          $Process->Suspend();
26.          sleep( 120 );
27.          while( $Process->Resume() > 1 ){ };
28.      }
29.      $Process->Kill();
30.  }
31. else
32. {
33.     print "Unable to start $Cmd.\n";
34.     print "Error: " .
Win32::FormatMessage( Win32::GetLastError() );
35. }

```

### ***Waiting for a Process to End***

At times, you will want to create a new process and then have your script wait until the new process terminates before continuing. This is why the `Wait()` method exists:

```
$Process->Wait( $TimeOut );
```

The first parameter (`$TimeOut`) is a timeout value. This value is in milliseconds and represents the length of time your script is willing to wait for the process to terminate. If it does not terminate by the determined time, your script continues processing regardless of whether the process has terminated. If this parameter is the constant `INFINITE`, your script will wait forever or until you power off your machine.

The `Wait()` method returns a `1` if the process has terminated while you were waiting; otherwise, it returns a `0`, meaning that the process is still running but the `Wait()` method has exhausted the timeout value.

## **Process Priority**

Because Win32 is a multitasking platform, the CPU (or CPUs if you have a multiprocessor box) runs one process for a specific duration of time and then moves on to run the next process. This continues until the CPU has spent a bit of time running each process, and then it repeats this pattern from the beginning of the process list. This way, each process runs just as much as the others, giving the illusion that the processes are all running simultaneously. At least, this is what you generally believe is happening.

In Win32, each process has a priority classification. This classification tells the operating system how responsive the process is, therefore clarifying how much attention to give it. Typically speaking, most processes are given a normal priority. Some processes might not need that much attention, however, so they can have a lower priority and require less attention from the CPU. A Web server needs to respond to incoming requests, for example, so it needs a fairly responsive class (like `NORMAL_PRIORITY_CLASS` or maybe even `HIGH_PRIORITY_CLASS`). A screen saver only needs to monitor whether the user has input anything before it kicks in. With such monitoring, there is no need for the screen saver to be very responsive; therefore, it could have the `IDLE_PRIORITY_CLASS` class.

Priorities are divided into four classes, as shown in [Table 8.2](#). After a process is set to a priority class, this generally tells you how responsive the process is. All new processes created without any specified priority class default to `NORMAL_PRIORITY_CLASS` with one exception: If a process that has a priority class of `IDLE_PRIORITY_CLASS` creates a child process, that child will also have a priority class of `IDLE_PRIORITY_CLASS`.

### **Getting a Process's Priority**

Retrieving the current priority class of a process is fairly simple; you use the `GetPriorityClass()` method of your `Win32::Process` object:

```
$Process->GetPriorityClass( $Priority );
```

The variable you pass in will be set to the actual priority class under which the process runs. This value will reflect one of the constants in [Table 8.5](#) later in this chapter. There is a trick, however, to obtaining the value. The value is returned as a packed character—that is, it is converted from the C long variable type to a Perl string. [Example 8.7](#) shows how to unpack the value. In the core distribution's `LibWin32` library and in the newer releases, this return value is already unpacked.

The `GetPriorityClass()` method will return a `1` if it successfully retrieved the priority class; otherwise, it will return a `0` to indicate that it failed.

[Example 8.7](#) shows how the `GetPriorityClass()` method is used. Notice that line 37 needs to exist only if you are not using `LibWin32` or Perl 5.005. Lines 22 and 26 both call a subroutine `GetPriority()`, which in turn calls `GetPriorityClass()` in line 34 and returns the priority class name. The process's priority is also changed in line 24.

#### **Example 8.7 Determining the priority class for a process**

```
01. use Win32::Process;
02. my %Class = (
03.     Win32::Process::IDLE_PRIORITY_CLASS => "IDLE" ,
```

```

04.     Win32::Process::NORMAL_PRIORITY_CLASS => "NORMAL",
05.     Win32::Process::HIGH_PRIORITY_CLASS => "HIGH",
06.     Win32::Process::REALTIME_PRIORITY_CLASS => "REALTIME"
07. );
08. my $App = "$ENV{WINDIR}\notepad.exe";
09. my $Cmd = "notepad readme.txt";
10. my $fInherit = 1;
11. my $Flags = 0;
12. my $Dir = ".";
13. my $Process;
14. my $fResult = Win32::Process::Create( $Process,
15.                                         $App,
16.                                         $Cmd,
17.                                         $fInherit,
18.                                         $Flags,
19.                                         $Dir);
20. if( $fResult )
21. {
22.     print "$Cmd has been created with a priority of ";
23.     print GetPriority( $Process ), "\n";
24.     print "Changing priority to IDLE_PRIORITY_CLASS...\n";
25.     $Process->SetPriorityClass( IDLE_PRIORITY_CLASS );
26.     print "Now process has a priority of ";
27.     print GetPriority( $Process ), "\n";
28. }
29.
30. sub GetPriority
31. {
32.     my( $Proc ) = @_;
33.     my $PriorityClass = "Unknown";
34.     my $Priority;
35.     if( $Proc->GetPriorityClass( $Priority ) )
36.     {
37.         if( $] < 5.005 )
38.         {
39.             # Next line not needed with libwin32 or Perl 5.005+
40.             $Priority = unpack( "c", $Priority );
41.         }
42.         $PriorityClass = $Class{$Priority};
43.     }
44.     return( $PriorityClass );
45. }

```

## **Setting a Process's Priority**

Just as you can retrieve a process's priority, you can also set it using the `SetPriorityClass()` method:

```
$Process->SetPriorityClass( $Priority );
```

The parameter you pass in (`$Priority`) is one of the constants listed in [Table 8.2](#).

If the `SetPriorityClass()` method is successful, it returns a `1`; otherwise, it returns `0` to indicate a failure.

Refer to [Example 8.7](#) and its explanation in the "Getting a Process's Priority" section.

### **Querying for a Process Handle**

Internally, Win32 manages processes by referring to a *process handle*. This is conceptually the same as a file handle because it simply is a value that refers to a running process. When Win32 creates a new process, it associates a handle with the process and returns it to the calling application.

The `Win32::Process` extension has a little-known secret function that will give you the process handle for a process created with the `Win32::Process::Create()` function:

```
$Process->get_process_handle();
```

Calling this method returns the numeric process handle for the process. This handle is useful if you want to call directly into the Win32 API. [Example 8.8](#) procures a process handle (line 28) so that it can call into the Win32 API's `GetProcessTimes()` function. This will return the start and end date stamp for the process. It also provides the amount of time the process spent in both the user and kernel modes (line 33). This script uses a bit of Win32 API magic to make all this work, but it illustrates how to use the `get_process_handle()`.

### **Example 8.8 Using a process handle**

```
01. use Win32::Process;
02. use Win32::API;
03. my $GetProcessTimes = new Win32::API( 'kernel32.dll',
04.                                         'GetProcessTimes',
05.                                         [N,P,P,P,P],
06.                                         I ) || die;
07. my $FileTimeToSystemTime = new Win32::API( 'kernel32.dll',
08.
09.                                         'FileTimeToSystemTime',
10.                                         [P,P],
11.                                         I ) || die;
12. my $App = "c:\\winnt\\notepad.exe";
13. my $Cmd = "notepad";
14. my $bInherit = 1;
15. my $Flags = CREATE_NEW_PROCESS;
16. my $Dir = "c:\\perl\\test";
17. my $Result = Win32::Process::Create( my $Process,
18.                                         $App,
19.                                         $Cmd,
20.                                         $bInherit,
21.                                         $Flags,
22.                                         $Dir);
23. if( $Result )
```

```

24.     my $pCreateTime = pack( "L2", 0, 0 );
25.     my $pExitTime = pack( "L2", 0, 0 );
26.     my $pKernelTime = pack( "L2", 0, 0 );
27.     my $pUserTime = pack( "L2", 0, 0 );
28.     my $hProcess = $Process->get_process_handle();
29.
30.     print "$Cmd has been created.\n";
31.     print "Its process ID is ", $Process->GetProcessID(), "\n";
32.     $Process->Wait( INFINITE );
33.     if( $GetProcessTimes->Call( $hProcess,
34.                                 $pCreateTime,
35.                                 $pExitTime,
36.                                 $pKernelTime,
37.                                 $pUserTime ) )
38.     {
39.         printf( "Time process spend in user mode: %s\n",
40.                 GetTime( $pUserTime ) );
41.         printf( "Time process spend in kernel mode: %s\n",
42.                 GetTime( $pKernelTime ) );
43.         printf( "Process started at: %s UTC\n",
44.                 GetTime( $pCreateTime ) );
45.         printf( "Process terminated at: %s UTC\n",
46.                 GetTime( $pExitTime ) );
47.     }
48. }
49. else
50. {
51.     print "The process $Cmd was not created!\n";
52.     print "Error: " .
Win32::FormatMessage( Win32::GetLastError() );
53. }
54.
55. sub GetTime
56. {
57.     my( $pFileTime ) = @_;
58.     my $Time = "???";
59.     my $pSystemTime = pack( "S8", 0, 0, 0, 0, 0, 0, 0, 0 );
60.     if( $FileTimeToSystemTime->Call( $pFileTime,
$pSystemTime ) )
61.     {
62.         my %Time;
63.         @Time{
64.             year, month, dow, day,
65.             hour, min, sec, msec } = ( unpack( "S*",
$pSystemTime ) );
66.         if( 1601 == $Time{year} )
67.         {
68.             # If the year == 1601 then we are referring to a
relative
69.             # amount of time
70.             $Time = sprintf( "%02d:%02d:%02d:%03d",
$Time{hour}, $Time{min},
71.                         $Time{sec} );

```

```

72.           $Time{sec}, $Time{msec} );
73.       }
74.   else
75.   {
76.       # We are dealing with a date/time stamp
77.       $Time = sprintf( "%02d/%02d/%04d %02d:%02d:%02d:%03d",
78.                           $Time{month}, $Time{day}, $Time{year},
79.                           $Time{hour}, $Time{min},
80.                           $Time{sec}, $Time{msec} );
81.   }
82. }
83. return( $Time );
84. }

```

## Creating a Process as Another User

There is yet another way to create a process: creating one as another user. The `Win32::AdminMisc` extension has a function that does just that, `CreateProcessAsUser()`. This will run an application using the security credentials of another user, providing the process with the security clearance granted to the specified user. This is how many services, such as Microsoft's Internet Information Service (IIS), provide user-specific permissions.

There are a few prerequisites to use this function:

- The user account that calls this function must have the following privileges (these can be assigned to a user in the User Manager program):

`SE_ASSIGNPRIMARYTOKEN_NAME` (Replace a process-level token)

`SE_INCREASE_QUOTA_NAME` (Increase quotas)

`SE_TCB_NAME` (Act as the operating system)

- A call to `Win32::AdminMisc::LogonAsUser()` (which is covered in [Chapter 9, "Console"](#)) must pre-cede this function; otherwise, the new process will be created using the current user.
- It can only be run on Windows NT\2000\XP because only these platforms support impersonation and creating processes while impersonating another user.

### Note

*If you are using Microsoft's Internet Information Server (IIS) 4.0, you will notice that the service uses an account such as `IUSER_<computername>` (where `<computername>` is your computer's network name) to anonymously access Web pages. This means that when a user tries to read a Web page without specifying a user ID and password, the Web server service tries to access the Web page using privileges given to `IUSR_<computername>`. If the Web page requested is a CGI script, the script will run under the `IUSER_<computername>` account. For more information on impersonation or logging on as another user, refer to [Chapter 9](#).*

If a user supplies a valid user ID and password when accessing a CGI script, the Web service will try to log on as the specified user. If the logon is successful, it will create the CGI script process using the same method that `CreateProcessAsUser()` uses.

You can choose an entire host of options if you want to enhance the process:

```
Win32::AdminMisc::CreateProcessAsUser( $Application [ ,  
$DefaultDirectory ] [ ,  
%Config ] );
```

The first parameter (`$Application`) is the application (and all parameters that need to be passed into it) that you want to run. (This is different from the `Win32::Spawn()` and `Win32::Process::Create()` functions.) If you wanted to run Notepad and have it load a file, for example, you could pass in the string "`Notepad.exe c:\\temp\\readme.txt`".

## Note

*The second and third parameters are optional. You can specify either one or both. It does not matter which one you use, but if you use both, the order is important.*

The second parameter (`$DefaultDirectory`), which is optional, is the default directory that the application will use.

The third parameter (`%Config`), which also is optional, is a hash of configuration information. Notice that this is not a reference to a hash but a hash itself. This means that you could, instead, replace the hash with a series of hash-like associations such as the following:

```
"XSize" => 200, "Ysize" => 50, ...
```

The configuration hash can contain keys and values listed in [Table 8.3](#). The key names in this hash are all case insensitive.

## Note

*You can use the extended abilities of AdminMisc's `CreateProcessAsUser()` without impersonating another user. If you do not call `LogonAsUser()` before calling the process creation function, then the new process is created running under the same account running the script.*

**Table 8.3. The Keys and Values of the %Config Hash When Using `Win32::AdminMisc::CreateProcessAsUser()`**

Key/Value	Description
-----------	-------------

**Table 8.3. The Keys and Values of the %Config Hash When Using `Win32::AdminMisc::CreateProcessAsUser()`**

Key/Value	Description
<code>Title</code>	The title of the process's window. This is what will be displayed in the title bar. This applies only to console-based applications because Windows applications are likely to overwrite this title with their own.
<code>Desktop</code>	A virtual desktop. Don't include this in your %Config hash unless you know what you are doing; otherwise, the process will attempt to be created in a "virtual" desktop. The default is " <code>winsta0\default</code> ".
<code>X</code>	The X coordinate of the upper-left corner of the process's window (in pixels).
<code>Y</code>	The Y coordinate of the upper-left corner of the process's window (in pixels).
<code>Xsize</code>	The width of the process's window (in pixels).
<code>Ysize</code>	The height of the process's window (in pixels).
<code>Xbuffer</code>	Number of chars wide that the buffer should be. This only applies to console-based applications.
<code>Ybuffer</code>	Number of chars high that the buffer should be. This also only applies to console-based applications.
<code>Fill</code>	The color to fill the window. This only applies to console-based applications. Table 8.6 lists possible values. These values can be logically OR'ed together.
<code>Priority</code>	The priority under which to run the process. It can use one of the constants defined in <a href="#">Table 8.2</a> .
<code>Flags</code>	Flags specifying process startup options. The possible values are defined in <a href="#">Table 8.1</a> and can be logically OR'ed together.
<code>ShowWindow</code>	State of the process's window during startup. <a href="#">Table 8.4</a> lists possible values.
<code>StdInput</code> , <code>StdOutput</code> , <code>StdError</code>	Specifies which handle to use for standard <code>STDIN</code> , <code>STDOUT</code> , and <code>STDERR</code> . If one of these is specified, all must be specified. You can use <code>Win32::AdminMisc::GetStdHandle()</code> to retrieve the current Win32 standard handles. Refer to the section "The STD Handles."
<code>Inherit</code>	Specifies to inherit Win32 file handles.
<code>Directory</code>	Specifies a default directory. This is the same attribute as the <code>\$DefaultDirectory</code> . You can use this instead of passing a string in as the second parameter to <code>CreateProcessAsUser()</code> .

[Table 8.4](#) lists the window state constants as specified by the `ShowWindow` key. Some of these are listed for the sake of Win32 API completeness but do not have any intrinsic value with current extension functions.

**Table 8.4. List of Window State Constants**

Constant	Description
<code>SW_HIDE</code>	The process is created but never shown because the window is hidden.
<code>SW_MAXIMIZE</code>	The process starts with its initial window maximized.
<code>SW_MINIMIZE</code>	The process starts with its initial window minimized.
<code>SW_RESTORE</code>	The process starts with its initial window in the state it was designed to be.
<code>SW_SHOW</code>	When the process is created, it is shown—that is, made visible. This was provided for completeness with the Win32 API and does not have any value with the <code>CreateProcessAsUser()</code> function because an application, when started, is typically visible to begin with.
<code>SW_SHOWDEFAULT</code>	Not applicable for the <code>CreateProcessAsUser()</code> function.
<code>SW_SHOWMAXIMIZED</code>	Same as <code>SW_MAXIMIZE</code> .
<code>SW_SHOWMINIMIZED</code>	Same as <code>SW_MINIMIZE</code> .
<code>SW_SHOWMINNOACTIVE</code>	The process is created in the minimized state, and the focus is not taken away from other processes.
<code>SW_SHOWNOACTIVATE</code>	The process is created, but it should not take the focus away from other processes.
<code>SW_SHOWNORMAL</code>	The process is displayed in the normal state.
<code>SW_SHOWNA</code>	The process is created in the default state. This is typically neither minimized nor maximized. However, the active window, before the process is created, remains active. The new process's window is not active.

The values in [Table 8.5](#), as specified by the `Fill` key, can be logically OR'ed together to make a composite color. For example, `BACKGROUND_RED | _BACKGROUND_BLUE | BACKGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY` will result in a white background (red and green and blue) with bright-blue characters (foreground).

**Table 8.5. Console Application Color Constants**

Constant	Description
<code>BACKGROUND_RED</code>	The color of the background will have a red component.
<code>BACKGROUND_BLUE</code>	The color of the background will have a blue component.
<code>BACKGROUND_GREEN</code>	The color of the background will have a green component.
<code>BACKGROUND_INTENSITY</code>	The background will be intensified (bright).
<code>FOREGROUND_RED</code>	The foreground text will have a red component.
<code>FOREGROUND_GREEN</code>	The foreground text will have a green component.
<code>FOREGROUND_BLUE</code>	The foreground text will have a blue component.
<code>FOREGROUND_INTENSITY</code>	The foreground text will be intensified (bright).

## Note

*Any process you run that requires use of the impersonated user's personal Registry settings will not function correctly. This is because the Win32 `LogonAsUser()` function that provides impersonation does not load the impersonated user's profile and Registry hive. If you need to access program settings and such, you should have your script load the Registry after you successfully impersonate the user. You can use the `Win32::Registry` extension to do this.*

## Case Study: Running Applications as Another User

Suppose you are an administrator, and you sometimes have to log on to your machine as another user to test for security permissions and such. You can log off of your workstation and log on as that user, but this would require you to close all your applications. Assume, for the sake of argument, that your work habits are like mine, and you have 50 or so windows open, all running different, seemingly important processes. This makes logging off a bit undesirable.

You decide to let Win32 Perl help you out of this dilemma, so you use a Perl script that will impersonate the user and run a program as that user. The script in [Example 8.9](#) provides this functionality.

### **Example 8.9 Running an application as another user**

```
01. use Win32::AdminMisc;
02. my( undef, $Domain, $User ) = ( $ARGV[0] ==~ 
/((.*?)\\)?(.*)$/ );
03. my $Password = $ARGV[1];
04. if( "*" eq $$Password )
05. {
06.     print "Enter Password:";
07.     $Password = <STDIN>;
08.     chop $Password;
09. };
10. my $Process = shift @ARGV || die;
11. print "\nStarting \"$Process\" as ", (" eq $$Domain) ?
$User:$Domain\\$User",
...\\n\\n";
12. if( Win32::AdminMisc::LogonAsUser($Domain,
13.                                     $User,
14.                                     $Password,
15.
LOGON32_LOGON_INTERACTIVE ) )
16. {
17.     my $LogonUser = Win32::AdminMisc::GetLogonName();
18.     print "Successfully logged on as $LogonUser.\n";
19.     print "\nLaunching $Process...\n";
20.     my $Pid = Win32::AdminMisc::CreateProcessAsUser(
21.                                         $Process,
22.                                         Flags    => CREATE_NEW_CONSOLE,
23.                                         XSize   => 640,
24.                                         YSize   => 400,
```

```

25.          X      => 200,
26.          Y      => 175,
27.          XBuffer => 80,
28.          YBuffer => 175,
29.          Show    => SW_MINIMIZE,
30.          Title   => "Title: $User" . " " . $Process
program",
31.          Fill    => BACKGROUND_BLUE | FOREGROUND_RED | FOREGROUND_BLUE | FOREGROUND_INTENSITY |
32.                                FOREGROUND_GREEN,
33.                                );
34.                                if( $Pid )
35.                                {
36.                                    print "Successful! The new processes PID is $Pid.\n";
37.                                }
38.                                else
39.                                {
40.                                    print "Failed.\n\tError: ", Error(), "\n";
41.                                }
42.                                }
43.                                else
44.                                {
45.                                    print "Failed to logon.\n\tError: ", Error(), "\n";
46.                                }
47.                                }
48.                                sub Error
49.                                {
50.                                    return Win32::FormatMessage( Win32::AdminMisc::GetError() );
51.                                }

```

To use the script in [Example 8.9](#), my account has to be granted the following privileges (their Win32 API equivalent names are also shown in parenthesis):

- The assign primary token privilege (`SE_ASSIGNPRIMARYTOKEN_NAME`)
- The increase quotas privilege (`SE_INCREASE_QUOTA_NAME`)
- The act as the operating system privilege (`SE_TCB_NAME`)

Assuming that the preceding script has the name runas.pl, you might typically run this command as:

```
perl runas.pl mydomain\JOEL JOELSPASSWORD "cmd"
```

or

```
perl runas.pl JOEL * "cmd"
```

Notice that in the first example the domain is specified, and in the second one no domain is specified (so it defaults to the primary domain). These examples will attempt to log on as `JOEL` with

the password `JOELSPASSWORD`. If the logon is successful, an attempt is made to run the `CMD` application. If all goes well, a DOS box will appear that has the security privileges of the user `JOEL`. You can access files that only that user account is allowed to access.

## Summary

Win32 processes can be quite simple to create, as with the `Win32::Spawn()` function, or as complex as `Win32::AdminMisc::CreateProcessAsUser()`. As these functions become more complex, so do their features and capabilities. It is important to note that, although each of the process-creating functions do practically the same thing, they provide radically different features.

The `Win32::Spawn()` function enables you to create a new process quickly and without much fuss. After the process is created, however, you cannot do much with it.

The `Win32::Process::Create()` function enables you to create a new process, but it takes a bit more understanding of how Win32 operates. The benefit, however, is that you can control the new process by using the various `Win32::Process` package's methods.

The `Win32::AdminMisc::CreateProcessAsUser()` function can be very easy to use or very complex, depending on what you want it to be. The benefit of this, of course, is that it runs processes as another user, which is quite beneficial at times.

# Chapter 9. Console, Sound, and Administrative Win32 Extensions

This chapter covers a few commonly used extensions that have remarkable practicality for an administrator:

- `Win32::Console`
- `Win32::Sound`
- `Win32::API`
- `Win32::AdminMisc`

## Console Windows

Win32 platforms support the concept of a *console*. Every Win32 Perl programmer is familiar with it; he just might not be aware of his familiarity. A console is a chunk of memory reserved by Win32 (called a *console buffer*) that is bound to a window that displays the contents of the buffer. A program can fill this chunk of memory with characters (and can specify the color of the characters). An application binds a console buffer to the console you see as the output text window of the program. When Perl executes a print command and you see the results on your screen, you are really seeing the output being sent to a console buffer and displayed by the console window. Consoles are created by requesting the Win32 OS to allocate one. Any application can request that a console be allocated for it, even GUI-based Windows applications (imagine something like MS Word with a DOS-like console window).

When an application allocates (creates) a console, three buffers are automatically created: one for `STDIN`, one for `STDOUT`, and one for `STDERR`. The buffer associated with `STDOUT` is displayed by

default. In the case of a DOS box or console-based application (like `perl.exe`), a console is automatically allocated, so there is no need for it to request that Win32 allocate the console.

An application (even a console-based one) can destroy its console and reallocate one later. No real benefit accrues by doing this, but it indeed can be done. An application can only have one console associated with it, so if it attempts to allocate more than one console, all requests after the first will fail.

This might sound just like a DOS box because every DOS box has a console and console buffer associated with it. This means that when you run a Perl program and a window appears for input and output, Win32 has automatically created a console buffer and bound input and output from the Perl script to it.

### Note

*It is important to know that an application can have any number of console buffers, but only one of them can be displayed at any given time. Additionally, you can change which buffer is displayed at any time. Displaying a different console buffer, however, does not change which buffer is considered to be the standard output device. This means that if you create a buffer and display it, all your print statements that are dumped to `STDOUT` will go to the original buffer, not to the currently displayed buffer.*

In these days of GUI environments, the Win32 console does not get much attention, but it can be remarkably powerful. Outputting data to a console is much easier than creating and managing GUI-based windows. Quick development of software can benefit from the `Win32::Console` extension.

All processes can have any number of console buffers associated with them, but only one can be displayed at any given time. This enables you to allocate several consoles to store information and then display whichever one is needed at a specific time. You could write a server application that accepts incoming named pipes or sockets, for example. If a message comes in from a computer belonging to upper management, it could be stored in the management buffer. User messages could be stored in the user buffer. Messages from the system (like computer error messages) could be stored in the system buffer. Your server process could enable you to switch between each of these consoles to be displayed.

### Tip

*Win32 defines two separate constants that represent the standard input and output devices:*

`CONIN$`  
`CONOUT$`

*These can both be used when opening a file. For example:*

```
open( INPUT, "< CONIN\$" ) || die;  
$In = <INPUT>;  
print "You typed '$In'\n";  
close( INPUT );
```

*I cannot think of a reason why you would want to use these instead of using STDIN and STDOUT, but nonetheless they exist.*

## Creating and Destroying a Console

If a Perl script needs to create a console, it can call the `Alloc()` function:

```
Win32::Console::Alloc();
```

The function takes no parameters, and if successful, it will allocate a console that can then display a console buffer. Generally speaking, a Perl script will never have a need to call this method because starting Perl itself automatically creates a console. However, if you are running a Perl script as a Win32 service, it would need to allocate a console window to display any output. Further discussion on writing Win32 services in Perl can be found in my book, *Win32 Perl Scripting: The Administrator's Handbook* (New Riders).

The `Alloc()` function will fail if the process already has a console allocated—a process can have only one console attached to it. If the method succeeds, it returns a `true` value (nonzero) and returns `false` (actually `undef`) if it fails.

### Tip

*When you create a process with the DETACHED\_PROCESS flag (see Chapter 8, "Processes"), it is created without any console allocated. If the process is a Perl script, it will have no way to output data to the screen. If it needs to have a console, you can always create one with the Alloc() method. If a script uses this, however, it should destroy the console with the Free() method before it terminates.*

If a console has been allocated (or created) with the `Alloc()` method, it must be destroyed (or freed) using the `Free()` function:

```
Win32::Console::Free();
```

The `Free()` function takes no parameters and will destroy the console attached to the process. A script can then reallocate a new console if need be.

Most Perl programs will never use this or the `Alloc()` method. But if one does, it can expect that a nonzero `true` value will be returned upon success in addition to destroying the console. A `false` value (`undef`) is returned upon error.

Both `Alloc()` and `Free()` can be run as methods of a console object, but it is not advised. Consoles are not related to buffers; instead, buffers are related to consoles. It is important to make this distinction between a buffer and a console. The console is just the window that displays the contents

of a console buffer. By destroying the console (the window), you are not destroying any of your buffers (except the `STDIN/STDOUT/STDERR` created automatically by the console).

`Alloc()` and `Free()` can be very handy if you are writing a Perl script that will run as a service that needs to display information. Services generally do not have consoles allocated for them by default. [Example 9.1](#) illustrates how a script can allocate a console and free it as well.

It is interesting to note that console buffers exist after you `Free()` your console and then `Alloc()` another one. It seems that they are not dependable, however. (It appears as if buffers start to overlap each other.) It is best to close all your buffers when you `Free()` your console and re-create them if you `Alloc()` another console. [Example 9.1](#) shows in line 14 that the buffer is `undef`'ed. This causes Perl to call the buffer object's `DESTROY()` method, which in turn causes the buffer to be freed. In this example, line 14 is not needed because the `$Buffer` object will fall out of scope (at the end of the script) and hence be automatically `undef`'ed.

### ***Example 9.1 Creating and destroying consoles***

```
01. use Win32::Console;
02. # Create a console buffer
03. my $Buffer = new Win32::Console();
04.
05. # This will fail if a console already exists.
06. Win32::Console::Alloc();
07. # Tell the OS to display the buffer in the console window.
08. $Buffer->Display();
09. $Buffer->Write( "This is being displayed in the console
window!\n" );
10. sleep(3);
11.
12. # Destroy any allocated console.
13. Win32::Console::Free();
14. undef $Buffer;
```

## **Creating Console Buffers**

Creating consoles is fairly easy; you can make a new `Win32::Console` object in two separate ways:

```
$Console = new Win32::Console( $Handle );
$Console = new Win32::Console( [ $$AccessMode [, $ShareMode ] ] );
```

In the first form, the only parameter is one of the handles listed in [Table 9.1](#). This is used to create an object that can be used to gather input or output to the current input and output devices.

***Table 9.1. The Standard Device Handles***

<b>Handle</b>	<b>Description</b>
<code>STD_OUTPUT_HANDLE</code>	The standard output device (typically the screen)
<code>STD_ERROR_HANDLE</code>	The standard error device (typically the screen)

**Table 9.1. The Standard Device Handles**

Handle	Description
STD_INPUT_HANDLE	The standard input device (typically the keyboard)

The second form requires two separate parameters. The first parameter (`$AccessMode`) specifies the access mode for the console and is optional. This mode determines how the console will be used: for input, for output, or both. This parameter can be a combination of the constants listed in [Table 9.2](#). These constants can be logically OR'ed together. If nothing is specified for this parameter, the default value of `GENERIC_READ | GENERIC_WRITE` will be used.

**Table 9.2. Console Access Mode Constants**

Constant	Description
<code>GENERIC_READ</code>	Provides read access to the console. This enables you to read data from the console.
<code>GENERIC_WRITE</code>	Provides write access to the console. This enables you to write data to the console.

The second parameter (`$ShareMode`) is optional and specifies the share mode of the console. This determines how your buffer can be accessed by other processes and Win32 file operations. [Table 9.3](#) lists the constants that can be used. These constants can be logically OR'ed together.

**Table 9.3. Console Share Mode Constants**

Constant	Description
<code>FILE_SHARE_READ</code>	Other processes can access the buffer for the purpose of reading data.
<code>FILE_SHARE_WRITE</code>	Other processes can access the buffer for the purpose of writing data to it.

Normally, sharing is not an issue with Perl scripts because they do not have access to the Win32 functions that would enable them to access such buffers from other processes. It is possible, however, that you could write a non-Perl application or a Perl extension that accesses these buffers. If this parameter is not supplied, it defaults to use `FILE_SHARE_READ | FILE_SHARE_WRITE`.

The `Console()` function is used either to create a new console buffer (the second form of the function) or to return the console buffer for a standard device (the first form of the function). [Example 9.2](#) shows how these buffers can be allocated and used. Just as in [Example 9.1](#), the last two lines (lines 15 and 16) are not necessary; they are only there to illustrate the proper way of closing the buffers. Because the end of the script causes the `$Buffer` and `$Out` objects to fall out of scope, they are automatically `undef`'ed.

### **Example 9.2 Creating and manipulating console buffers**

```
01. use Win32::Console;
```

```

02. my $Out = new Win32::Console( STD_OUTPUT_HANDLE ) || die;
03. my $Buffer = new Win32::Console() || die;
04. $Buffer->Write( "This is my nifty display buffer\n" );
05. $Buffer->Display();
06. $Buffer->Write( "We will count to 10 then go back to the
original display.\n" );
07. for my $Count ( 1 .. 10 )
08. {
09.     $Buffer->Write( "$Count\n" );
10. }
11. $Buffer->Write( "Hit the enter key to continue...\n" );
12. <STDIN>;
13. $Out->Display();
14. print "Welcome back to the original display!\n";
15. undef $Buffer;
16. undef $Out;

```

Notice that in [Example 9.2](#), line 3 creates a console buffer object for a new buffer (`$Buffer`), and line 2 creates a console buffer object for the standard output device (`$Out`). The only reason for making the `$Out` object is so we can display it in the end of the script. Otherwise, there would be no way to display the standard output buffer.

If the new `Win32::Console()` call is successful, it returns a Perl object that represents the particular console buffer. If the function fails, it returns a `FALSE` value (to be exact). When a buffer is no longer needed, it is best to destroy it with the `undef` command so that it is removed from memory:

```
undef $Buffer;
```

Alternatively, you can use the `_CloseHandle()` method to force the buffer to be purged from memory without undefining the variable:

```
$Buffer->_CloseHandle();
```

This method takes no parameters and is undocumented, but this is what is called during a `DESTROY()` call on the object (which occurs when you `undef` the object). It is best that you use `undef`, but if you want to experiment, the `_CloseHandle()` does exist. Keep in mind that because it is undocumented, it might change in some future version of the extension.

## Displaying a Console Buffer

Suppose you have created a dozen console buffers and have been writing things into them, but you still only see the `STDOUT` in your console window. What good are these buffers? Well, here comes the fun part: You can tell the console window to display whichever buffer you want by using the `Display()` method:

```
$Buffer->Display();
```

The `Display()` method takes no parameters and will return a `TRUE` value (nonzero) if it successfully switches to the new buffer; otherwise, it will return `FALSE` (`undef`).

### Tip

*You will want to know about a little trick when using this method. After you have displayed a buffer, the console will continue displaying that buffer until told otherwise. There is no automatic way to switch back to the `STDOUT` buffer. So, before you use `Display()`, you will probably need to create a console object based on the `STD_OUTPUT_HANDLE` constant (refer to [Example 9.2](#)).*

## Writing to a Console Buffer

After a console buffer has been either allocated or obtained (in the case of the standard input/output/error handles), text can be written to it. It is important to remember that `STDOUT` is really just a console buffer. When you print something to `STDOUT`, it is being printed to the `STDOUT` console buffer. [Example 9.3](#) shows three different ways to perform the same function of printing to `STDOUT`. This example is a bit extreme because there is quite a bit of overhead just to print to `STDOUT` by means of writing directly to its console buffer, but it demonstrates how it can be accomplished.

### Example 9.3 Various methods of printing to `STDOUT`

```
01. use Win32::Console;
02. print "This is attempt 1 to print to STDOUT.\n";
03. print STDOUT "This is attempt 2 to print to STDOUT.\n";
04. my $Out = new Win32::Console( STD_OUTPUT_HANDLE ) || die;
05. $Out->Write( "This is attempt 3 to print to STDOUT.\n" );
06. undef $Out;
```

## Clearing a Console Buffer

Before learning about writing data, you first need to understand how to clear a buffer. If a buffer has become cluttered, you can clear the contents of it by using the `cls()` method:

```
$Buffer->Cls( [ $$ColorAttribute ] );
```

The only parameter (`$ColorAttribute`) is optional, and it specifies the color with which to fill the buffer. This value can be any combination of values documented later in this chapter in [Table 9.8](#) (or one foreground and background color variable from [Table 9.9](#)). This combination of values is logically OR'ed together. The net effect of calling the `Cls()` method with a color attribute parameter is to clear the buffer and change the color of it at the same time.

If no parameter is passed into the method, the value of the `$main::ATTR_NORMAL` variable will be used. By default, the extension defines this variable as gray text on a black background.

If the `Cls()` method is successful, it fills the buffer with the space character (and thus "clears" the buffer), and each character's color attribute will be set to either `$main::ATTR_NORMAL` or whatever is passed into the method. If the method fails, `undef` is returned.

### **Writing to a Console Buffer with Write()**

Writing to a console buffer is performed by using the `Write()` method:

```
$Buffer->Write( $Data );
```

The only parameter is any text string to be written to the console buffer. You could specify this as binary data if you like, but because it is to be dumped into a text-based console buffer, it will result in the appropriate ASCII characters being displayed. This poses a problem for character data that you need to write that are ANSI-based (such as accented characters or umlauts). In situations such as these, you need to convert the ANSI character to a corresponding OEM character.

If the `Write()` method is successful, it will return the number of characters that were written into the buffer; otherwise, it will return `undef`.

The text will start to be displayed at the location of the buffer's cursor. You can change the cursor's position prior to using the `Write()` method.

The text will be printed using the color attributes associated with the buffer. Refer to the `Attr()` method for more information regarding this. Refer to [Example 9.3](#) for a demonstration of the `Write()` method.

### **Writing to a Console Buffer with WriteChar()**

Another way to write data to a buffer is to do so by specifying at which `x` and `y` location to start writing. You use the `WriteChar()` method for this:

```
$Buffer->WriteChar( $Data, $X, $Y );
```

The first parameter (`$Data`) is the text data to write into the buffer. This can be data that was returned from the `ReadChar()` method; see the section "[Reading from a Console Buffer](#)" later in this chapter for details. The second and third parameters (`$X` and `$Y`) are the X and Y coordinates, respectively, of the character location where you want the data to begin writing.

The `WriteChar()` method is not quite what you might believe it to be. The data is written to the buffer without using any of the color attributes associated with the buffer. In other words, whatever text colors exist that your data will overwrite will remain after the `WriteChar()` method has completed. If you want to also change the colors, you must execute the `WriteAttr()` method.

If the `WriteChar()` method is successful, it returns the number of characters written; otherwise, it returns `undef`. See [Example 9.6](#) in the next section for a demonstration of the `WriteChar()` method.

### **Applying Colors to a Console Buffer with WriteAttr()**

Because the `WriteChar()` method does not write color attributes for each character, another method performs that task: `WriteAttr()`.

```
$Buffer->WriteAttr( $Data, $X, $Y );
```

The first parameter (`$Data`) is a string of color attributes (one attribute per character) that will be written. If you supply a string of three color attributes, only three characters will be colored. Each color attribute consists of any combination of constants documented later in this chapter in [Table 9.8](#) (or any foreground and background variable combination from [Table 9.9](#)), logically OR'ed together. The resulting value must be a binary value, not an interpreted value. This means that when you create the string, you must convert each resulting color value into a discrete character. You can do this by using the Perl `chr()` or `pack()` functions, as demonstrated in [Example 9.4](#) and [Example 9.5](#). Even though both examples use different techniques, they are equivalent.

The second and third parameters for `WriteAttr()` (`$X` and `$Y`) are the X and Y coordinates, respectively, where the data will begin to be written in the buffer. This is relative to position `0,0` being in the upper-left corner of the buffer.

#### **Example 9.4 Creating a data string for the `WriteAttr()` method using `chr()`**

```
$Data = chr( $FG_YELLOW | $BK_BLUE );
```

#### **Example 9.5 Creating a data string for the `WriteAttr()` method using `pack()`**

```
$Data = pack( "c", ( $FG_YELLOW | $BK_BLUE ) );
```

If the `WriteAttr()` method is successful, it will write the color data into the buffer at the specified coordinates and then return the number of attributes that are written. If it fails, `WriteAttr()` will return `undef`.

[Example 9.6](#) demonstrates the use of the `WriteAttr()` method. Lines 28 and 52 read attribute and character data from the first buffer. (These read methods are covered later in this chapter in the section "[Reading from a Console Buffer](#).") Line 31 uses the `WriteChar()` method to write the color attributes to the new buffer. Line 55 writes the character data to the new buffer using the `WriteAttr()` method.

## Bug Report

[Example 9.6](#) might end up failing on your machine. This is because some versions of Win32::Console have a bug in their `ReadAttr()` methods. See the section "["Retrieving Color Data from a Console Buffer"](#)" later in this chapter for details on how to correct this bug.

### **Example 9.6 Copying character and color data from one buffer to another using `WriteChar()` and `WriteAttr()`**

```
01. use Win32::Console;
02. my $Buf1 = new Win32::Console();
03. my $Buf2 = new Win32::Console();
04. my $StdOut = new Win32::Console( STD_OUTPUT_HANDLE );
05. # Set the colors for Buffer 1
06. $Buf1->Attr( $FG_YELLOW | $BG_BLUE );
07. # Clear both buffers
08. $Buf1->Cls();
09. $Buf2->Cls( $FG_WHITE | $BG_GREEN );
10. # Write a bunch of stuff to buffer 1
11. $Buf1->Write( join("", (a..z) x 50) ) );
12. $Buf1->Write( "\n\n" );
13. print "Let's look at the first buffer...\n";
14. print "Hit ENTER to see it\n";
15. <STDIN>;
16. Show( $Buf1 );
17. $StdOut->Display();
18.
19. print "Now let's look at the second (empty) buffer...\n";
20. print "Hit ENTER to see it\n";
21. <STDIN>;
22. Show( $Buf2 );
23. $StdOut->Display();
24.
25. print "Now we are reading a line of color from Buffer 1\n";
26. print "and writing it to Buffer 2...\n";
27. # Let's copy the color attributes...
28. if( my $Data = $Buf1->ReadAttr( 60, 10, 12 ) )
29. {
30.     print "Writing color information into Buffer 2...";
31.     if( ! $Buf2->WriteAttr( $Data, 10, 12 ) )
32.     {
33.         print "Failure!\n";
34.     }
35.     else
36.     {
37.         print "Success!\n";
38.     }
39. }
40. else
41. {
42.     print "Reading data failed.\n";
43. }
```

```

44. print "Hit ENTER to see it\n";
45. <STDIN>;
46. Show( $Buf2 );
47. $StdOut->Display();
48.
49. print "Now we are reading a line of text from Buffer 1\n";
50. print "and writing it to Buffer 2...\\n";
51. # Read a rectangle of data from Buffer 1
52. if( my $Data = $Buf1->ReadChar( 60, 10, 12 ) )
53. {
54.     print "Writing character data to Buffer 2...";
55.     if( ! $Buf2->WriteChar( $Data, 10, 12 ) )
56.     {
57.         print "Failure!\\n";
58.     }
59.     else
60.     {
61.         print "Success!\\n";
62.     }
63. }
64. else
65. {
66.     print "Reading data failed.\\n";
67. }
68. print "Hit ENTER to see it\\n";
69. <STDIN>;
70.
71. Show( $Buf2 );
72. $StdOut->Display();
73.
74. sub Show
75. {
76.     my( $Buffer ) = @_;
77.     $Buffer->Display();
78.     $Buffer->Cursor( 0, 22 );
79.     $Buffer->Write( "Hit ENTER to continue..." );
80.     $In = <STDIN>;
81. }

```

### **Writing Blocks of Data to a Console Buffer**

Another way to write data to a buffer is by specifying a rectangle of character and color data. The method that does this is `WriteRect()`:

```
@Result = $Buffer->WriteRect( $Rect, $Left, $Top, $Right,
$Bottom );
```

The parameters for the `WriteRect()` method are as follows:

- `$Rect` is binary data that is the result of a call to the `ReadRect()` method.

- `$Left` is the column number of the left side of the rectangle.
- `$Top` is the row number of the top of the rectangle.
- `$Right` is the column number of the right side of the rectangle.
- `$Bottom` is the row number of the bottom of the rectangle.

If the `WriteRect()` method is successful, the data overwrites whatever is in the area that the rectangle covers, and an array is returned representing the coordinates of the rectangle that was written. See [Table 9.4](#) for details on the returned array. If the method fails, `undef` is returned.

**Table 9.4. The Array Returned from the `WriteRect()` Method**

Array Element	Description
<code>\$Result[0]</code>	The column number of the left side of the rectangle
<code>\$Result[1]</code>	The row number of the top of the rectangle
<code>\$Result[2]</code>	The column number of the right side of the rectangle
<code>\$Result[3]</code>	The row number of the bottom of the rectangle

[Example 9.7](#) demonstrates use of the `WriteRect()` method.

### **Example 9.7 Copying a block from one buffer to another using `WriteRect()`**

```

01. use Win32::Console;
02. my @Rect = ( 2, 3, 60, 20 );
03. my $Buf1 = new Win32::Console();
04. my $Buf2 = new Win32::Console();
05. my $StdOut = new Win32::Console( STD_OUTPUT_HANDLE );
06. # Set the colors for Buffer 1
07. $Buf1->Attr( $FG_YELLOW | $BG_BLUE );
08. # Clear both buffers
09. $Buf1->Cls();
10. $Buf2->Cls( $FG_WHITE | $BG_GREEN );
11. # Write a bunch of stuff to buffer 1
12. $Buf1->Write( join('', (a..z) x 50 ) );
13. $Buf1->Write( "\n\n" );
14. print "Let's look at the first buffer...\n";
15. print "Hit ENTER to see it\n";
16. <STDIN>;
17. Show( $Buf1 );
18. $StdOut->Display();
19.
20. print "Now let's look at the second (empty) buffer...\n";
21. print "Hit ENTER to see it\n";
22. <STDIN>;
23. Show( $Buf2 );
24. $StdOut->Display();
25.
26. print "Now let's copy a rectangle from Buffer 1 to Buffer
2...\\n";

```

```

27. # Read a rectangle of data from Buffer 1
28. if( $Rect = $Buf1->ReadRect( @Rect ) )
29. {
30.     print "Writing data to Buffer 2...";
31.     if( ! $Buf2->WriteRect( $Rect, @Rect ) )
32.     {
33.         print "Failure!\n";
34.     }
35.     else
36.     {
37.         print "Success!\n";
38.     }
39. }
40. else
41. {
42.     print "Reading rectangle failed.\n";
43. }
44.
45. print "Now let's look at Buffer 2 again...\n";
46. print "Hit ENTER to see it\n";
47. <STDIN>;
48. Show( $Buf2 );
49. $StdOut->Display();
50.
51. sub Show
52. {
53.     my( $Buffer ) = @_;
54.     $Buffer->Display();
55.     $Buffer->Cursor( 0, 22 );
56.     $Buffer->Write( "Hit ENTER to continue..." );
57.     $In = <STDIN>;
58. }

```

### **Scrolling Blocks of Data Around in a Console Buffer**

You can use one final method to write data to a buffer. Actually, it moves data from one point in a buffer to another point within the same buffer. This is the `Scroll()` method:

```
$Buffer->Scroll( $Left, $Top, $Right, $Bottom, $Col, $Row, $Char,
$Attribute [,
$ClipLeft, $ClipTop, $ClipRight, $ClipBottom ] );
```

The parameters for the `Scroll()` method are as follows:

- `$Left` is the column number of the left side of the rectangle.
- `$Top` is the row number of the top of the rectangle.
- `$Right` is the column number of the right side of the rectangle.
- `$Bottom` is the row number of the bottom of the rectangle.
- `$Col` and `$Row` are the X (column) and Y (row) coordinates (respectively) to which the upper-left corner of the rectangle defined by the first four parameters will be moved.

- `$Char` is the character that will be used to fill the space that has been moved. This value must be an integer, not a character or character string. Do not use some value like "A" or 'A'; instead, use a value like that returned by `unpack("c", "A")`.
- `$Attribute` is the color attribute of the character that will be used to fill the space that has been moved.
- `$ClipLeft`, `$ClipTop`, `$ClipRight`, and `$ClipBottom` are the left, top, right, and bottom row and column values (respectively) that define a clipping region.

The documentation claims that these parameters are optional, and technically they are. However, a bug in the extension prevents them from being optional unless you modify the package file. See the note later in this section for more details.

If the `Scroll()` method is successful, it returns a nonzero (True) value and moves the contents of the defined rectangle to its new upper-left coordinates defined by the fifth and sixth parameters. The space left empty after the move will be filled with the character and color defined by the seventh and eighth parameters. If the `Scroll( )` method fails, it returns `undef`.

Specifying a clipping rectangle (parameters 9 through 12) will prevent any of the moved rectangle from moving outside the boundaries specified by the clipping rectangle. This protects the contents of the buffer outside the clipping rectangle from being overwritten. [Example 9.8](#) demonstrates use of the `Scroll()` method.

It is interesting to note that lines 20 through 25 use `Scroll()` to move a block of data from one location to another within the buffer without clipping. Well, there is clipping, but the clipping rectangle is so large that clipping won't occur. The first parameter is an array of four values (the position of the block to be moved). Also notice that "fill character" is the result of an `unpack()` function.

Lines 40 through 45 are practically the same as lines 20 through 25 except that a clipping window is specified (the last four parameters) by an array of four values. If the any part of the block of data that is moved (or scrolled) falls out of the clipping window, that data is discarded. This is very practical if you need to make sure that any size of block is moved into a smaller window without overwriting the window border.

### **Example 9.8 Using the `Scroll()` method both with and without a clipping region**

```

01. use Win32;
02. use Win32::Console;
03. my $FillChar = " ";
04. my $FillColor = $FG_YELLOW | $BG_BLACK;
05. my @Rect = ( 2, 2, 50, 10 );
06. my @Rect2 = ( 2, 2, 50, 10 );
07. my @Clip = ( 0, 0, 50, 10 );
08. my $iCount = 14;
09. # Create our console buffers
10. my $StdOut = new Win32::Console( STD_OUTPUT_HANDLE );
11. my $Buffer = new Win32::Console();
12. $Buffer->Display();
13. # Fill the buffer with nonsense...
14. FillBuffer( $Buffer );
15. # Now let's scroll a block...
16.

```

```

17. while( $iCount-- )
18. {
19.     map{ $Rect[$_] += 1; } (0..3);
20.     if( ! $Buffer->Scroll( @Rect,
21.                             $Rect[0] ++ 1,
22.                             $Rect[1] ++ 1,
23.                             unpack("c", $FillChar),
24.                             $FillColor,
25.                             0, 0, 10000, 10000 ) )
26.     {
27.         Write( $Buffer, "Scroll failed." );
28.         last;
29.     }
30.     Win32::Sleep( 50 );
31. }
32.
33. Write( $Buffer, "Hit ENTER to end." );
34. $In = <STDIN>;
35. FillBuffer( $Buffer );
36. $iCount = 14;
37. while( $iCount-- )
38. {
39.     map{ $Rect2[$_] += 1; } (0..3);
40.     if( ! $Buffer->Scroll( @Rect2,
41.                             $Rect2[0] ++ 1,
42.                             $Rect2[1] ++ 1,
43.                             unpack("c", $FillChar),
44.                             $FillColor,
45.                             @Clip ) )
46.     {
47.         Write( $Buffer, "Scroll failed." );
48.         last;
49.     }
50.     Win32::Sleep( 50 );
51. }
52. Write( $Buffer, "Hit ENTER to end." );
53. $In = <STDIN>;
54.
55. sub FillBuffer
56. {
57.     my( $Buffer ) = @_;
58.     my $Char = "a";
59.     my $Count = 20;
60.     #clear and set the buffer's color
61.     $Buffer->Attr( $FG_YELLOW | $BG_BLUE );
62.     $Buffer->Cls();
63.     # Fill the buffer full of just nonsense stuff
64.
65.     while( $Count-- )
66.     {
67.         $Buffer->Write( $Char x 75 . "\n" );
68.         $Char++;

```

```

69.    }
70. }
71.
72. sub Write
73. {
74.     my( $Buffer, $String ) = @_;
75.     my $Color = chr( $FG_WHITE | $BG_BLUE ) x length( $String );
76.     $Buffer->Write("\n\n");
77.     my( $x, $y ) = $Buffer->Cursor();
78.     $Buffer->WriteChar( $String, $x, $y );
79.     $Buffer->WriteAttr( $Color, $x, $y );
80. }

```

## Note

*Regardless of what the documentation says, a bug in Win32::Console forces you to always pass in a clipping range. Even if you don't want any clipping, you need to specify the range.*

*The way around this bug is to specify a large clipping range, one so large that it would not impact any display regardless of size. A good sample clipping range is (0,0,10000,10000).*

*You can patch the CONSOLE.PM file if you don't want to always have to specify a clipping size. To do this, you need to locate the Scroll subroutine in the file. Add the following lines just before the first return statement:*

```

my( @Clip ) = ( 0, 0, 10000, 10000 );
@Clip = ( $left2, $tip2, $right2, $bottom2 ) if defined $left2;

```

*The next alteration is that you need to replace the last four parameters passed into the \_ScrollConsoleScreenBuffer() function with @Clip. This would make the function look like the following:*

```

return _ScrollConsoleScreenBuffer($self->{ 'handle' }, $left1, $top1,
$right1, $bot
tom1, $col, $row, $char, $attr, @Clip );

```

*This fix is applicable to Win32::Console version 0.03 (07 Apr 1997) and later.*

## Reading from a Console Buffer

Just as you can write to a console buffer, you can read from a buffer. You can read characters, the color attributes, or a rectangle of both. The `ReadChar()` method will read a number of characters from the buffer:

```
$Data = $Buffer->ReadChar( $Number, $X, $Y );
```

The first parameter (`$Number`) is the number of characters to read. This must be a positive integer.

The second and third parameters (`$X` and `$Y`) are the column (the X coordinate) and the row (the Y coordinate), respectively, that indicate where to begin reading within the buffer.

If the `ReadChar()` method is successful, it returns a string of character data; otherwise, it returns `undef`.

### **Retrieving Color Data from a Console Buffer**

A related method does practically the same thing as `ReadChar()` except it reads the color attributes rather than the character data. This is the `ReadAttr()` method:

```
$Data = $Buffer->ReadAttr( $Number, $X, $Y );
```

The first parameter (`$Number`) is the number of characters to read. This must be a positive integer. The second and third parameters (`$X` and `$Y`) are the column (the X coordinate) and the row (the Y coordinate), respectively, that indicate where to begin reading within the buffer.

The results of the `ReadAttr()` method (if it is successful) can be applied using the `WriteAttr()` method. If the `ReadAttr()` is successful, it returns a string of character data; otherwise, it returns `undef`.

## Bug Report

*A bug has crept into newer versions of the `Win32::Console` extension. When you call in to the `ReadAttr()` method, the script will croak and print out something similar to:*

[\[View full width\]](#)

```
Usage: Win32::Console::_ReadConsoleOutputAttribute(handle,
charbuf, len, x, y) at
c:/Perl/site/lib/Win32/Console.pm line 189, <STDIN> line 4.
```

*The problem is that the `Console.pm` file forgot to pass a buffer (`charbuf`) into the `_ReadConsoleOutputAttribute()` function. But don't fret; this is easily enough fixed. Just open the `Console.pm` file and modify the `ReadAttr()` method to look like this:*

[\[View full width\]](#)

```
#=====
# sub ReadAttr {
# =====
my($self, $size, $col, $row) = @_;
my $Buffer;
return undef unless ref($self);
return _ReadConsoleOutputAttribute($self->{ 'handle' },
$Buffer,
```

```
    $size, $col, $row);  
}
```

The only changes are that we added line 5 and inserted `$Buffer` into the second parameter on line 7.

### **Reading Blocks of Data from a Console Buffer**

The last of the reading methods will read a rectangular area of both character and color data. This is the `ReadRect()` method:

```
$Data = $Buffer->ReadRect( $Left, $Top, $Right, $Bottom );
```

The parameters for the `ReadRect()` method are defined as follows:

- `$Left` is the column number of the left side of the rectangle.
- `$Top` is the row number of the top of the rectangle.
- `$Right` is the column number of the right side of the rectangle.
- `$Bottom` is the row number of the bottom of the rectangle.

If the `ReadRect()` method is successful, the character and color data that represents the area of the rectangle is returned in a binary format. This can be used in conjunction with the `WriteRect()` method. If the method fails, `undef` is returned.

### **Reading User Input from a Console Buffer**

Another method reads data from a buffer, but this only works for a console buffer object created by specifying the `STD_INPUT_HANDLE` constant as the only parameter to the new function. Such a console object can be used to execute the `InputChar()` method:

```
$Data = $InputBuffer->InputChar( $Number );
```

The first parameter (`$Number`) is optional and is a numeric integer value that represents the total number of characters to read. If no value is specified, it defaults to 1.

The `InputChar()` method returns the characters that have been read. The method does not return until `$Number` of characters has been read. If the `InputChar()` method fails, it returns `undef`. [Example 9.9](#) demonstrates how `InputChar()` is used.

In this example, the `Mode()` method is called to set various input options (line 3). This method is discussed later in this chapter in the section "[Setting I/O Modes for Console Windows](#)."

Line 5 only accepts one character. The rest of the code compares that character value to see if the user selected to quit or continue.

#### **[Example 9.9 Using the `InputChar\(\)` method](#)**

```

01. use Win32::Console;
02. my $StdIn = new Win32::Console( STD_INPUT_HANDLE );
03. $StdIn->Mode( ENABLE_PROCESSED_INPUT );
04. print "Hit 'C' to continue or 'Q' to Quit....\n";
05. while( my $Data = $StdIn->InputChar( 1 ) )
06. {
07.     if( lc $Data eq "q" )
08.     {
09.         print "Quitting...\n";
10.         exit;
11.     }
12.     elsif( lc $Data eq "c" )
13.     {
14.         last;
15.     }
16. }
17. print "Continuing...\n";
18. # ...add your code...

```

## Console Properties

General attribute information regarding the console window can be obtained by using the `Info()` method. This method can only be called from a noninput control buffer:

```
@Attributes = $Buffer->Info();
```

The method takes no parameters and returns an array of values described in [Table 9.5](#). If the `Info()` method is successful, an array is returned; otherwise, `undef` is returned.

**Table 9.5. Array Elements Returned by the `Info()` Method**

Array Element	Description
<code>\$Attributes[0]</code>	The number of columns in the current console buffer.
<code>\$Attributes[1]</code>	The number of rows in the current console buffer.
<code>\$Attributes[2]</code>	The column where the cursor is located in the current console buffer.
<code>\$Attributes[3]</code>	The row where the cursor is located in the current console buffer.
<code>\$Attributes[4]</code>	The color attribute that will be used to write data in the current console buffer.
<code>\$Attributes[5]</code>	The column number of the current buffer displayed as the left side of the current console window.
<code>\$Attributes[6]</code>	The row number of the current buffer displayed as the top line of the current console window.
<code>\$Attributes[7]</code>	The column number of the current buffer displayed as the right side of the current console window.

**Table 9.5. Array Elements Returned by the `Info()` Method**

Array Element	Description
<code>\$Attributes[8]</code>	The row number of the current buffer displayed as the last line of the current console window.
<code>\$Attributes[9]</code>	The maximum number of columns that the current console window can display. This value depends on the current buffer size, font, and other such values.
<code>\$Attributes[10]</code>	The maximum number of rows that the current console window can display. This value depends on the current buffer size, font, and other such values.

### **Finding the Size of a Console Window**

To discover what the largest possible console window size is, use the `MaxWindow()` method:

```
( $MaxCol, $MaxRow ) = $Buffer->MaxWindow( [$Flag] );
```

The only parameter (`$Flag`) is optional. If this parameter is passed in, then the method returns the largest number of rows and columns that the buffer can be configured to be and still be visible in the display with the current font and at the current screen resolution. This is useful if you want to determine how big your screen size *can* be so you can modify the buffer settings.

If the optional parameter is not passed in, then the method returns the size that the console window was set to when the console buffer was created. This is the size the window will become if you maximize it.

## **Max Size After a Manual Resize**

*If you resize the console window manually (by using the mouse or keyboard and selecting the Properties option in the console window's system menu), the new maximum size is not reflected in a call to `MaxWindow()`.*

If the `MaxWindow()` method is successful, it returns an array consisting of the highest number of columns and rows possible for a console window. If `MaxWindow()` fails, it returns `undef`.

### **Setting I/O Modes for Console Windows**

A console window can have particular modes set for input and output. These modes can be set using the `Mode()` method:

```
$Buffer->Mode( [ $$Mode ] );
```

The first parameter (`$Mode`) is optional and can consist of any combination of the constants listed in [Table 9.6](#). These constants can be logically OR'ed together.

The `Mode()` method returns the current mode of the console window (and displayed console buffer). If the method fails, it returns `undef`.

By default, all consoles are created with all of the input options in [Table 9.6](#) set except for `ENABLE_WINDOW_INPUT`.

**Table 9.6. Console Window Modes**

Mode	Description
<code>ENABLE_ECHO_INPUT</code>	Input is echoed to the current console buffer. This mode can be used only if the <code>ENABLE_LINE_INPUT</code> mode is also enabled.
<code>ENABLE_LINE_INPUT</code>	This will accept input only after a carriage return is supplied. If this flag is not set, input can be read one character at a time. Refer to the <code>InputChar()</code> method.
<code>ENABLE_MOUSE_INPUT</code>	If the mouse pointer is within the borders of the console window and the window has the keyboard focus, mouse events generated by mouse movement and button presses are placed in the input buffer. To use this, you need to first disable Quick Edit mode in the Console Properties dialog box.
<code>ENABLE_PROCESSED_INPUT</code>	Ctrl+C is processed by the system and is not placed in the input buffer. This means the Perl script will not be able to see the Ctrl+C event. Other control keys are also processed by the system and are not placed into the input buffer. If the <code>ENABLE_LINE_INPUT</code> mode is also enabled, backspace, carriage return, and linefeed characters are handled by the operating system and are not placed into the input buffer.
<code>ENABLE_PROCESSED_OUTPUT</code>	Characters written to the console buffer using the <code>Write()</code> method are examined for ASCII control sequences, and the correct action is performed. Backspace, tab, bell, carriage return, and linefeed characters are processed.
<code>ENABLE_WINDOW_INPUT</code>	User interactions that change the size of the console screen buffer are reported in the console's input buffer. Information about these events can be read from the input buffer.
<code>ENABLE_WRAP_AT_EOL_OUTPUT</code>	When writing to the buffer, the cursor moves to the beginning of the next row when it reaches the end of the current row. This causes the rows displayed in the console window to scroll up automatically when the cursor advances beyond the last row in the window. It also causes the contents of the screen buffer to scroll up (discarding the top row of the screen buffer) when the cursor advances beyond the last row in the screen buffer. If this mode is disabled, the last character in the row is overwritten with any subsequent characters.

Example 9.10 illustrates how the `Mode()` method can be used. In this example, the user is prompted for a password. For every character that is typed, a \* character is echoed to the screen. Line 8 checks to see if the user hit the Enter key.

Line 12 checks for the Backspace key. Notice when Backspace is hit, the script will print a backspace followed by a space and another backspace. This effectively erases the last character from the screen. The `$Password` variable is "chopped" as well, removing the last character from the string. If there are no characters, `chop()` returns an empty string.

### **Example 9.10 Entering a password**

```
01. use Win32::Console;
02. my $StdIn = new Win32::Console( STD_INPUT_HANDLE );
03. my $Password = "";
04. $StdIn->Mode( ENABLE_PROCESSED_INPUT );
05. print "Enter password: ";
06. while( my $Data = $StdIn->InputChar( 1 ) )
07. {
08.     if( "\r" eq $$Data )
09.     {
10.         last;
11.     }
12.     elsif( "\ch" eq $$Data )
13.     {
14.         if( "" ne chop(( $Password ) )
15.         {
16.             print "\ch \ch";
17.         }
18.         next;
19.     }
20.     $Password .= $Data;
21.     print "*";
22. }
23. print "\nYour password is: '$Password'\n";
```

### **Reading and Setting Titles for Console Windows**

The title of the console window can be both read and set using the `Title()` method:

```
$Buffer->Title( [$Title] );
```

The only parameter (`$Title`) is optional and can be any text string. There is no limit to the length of the string.

The method returns the current title of the console window. The title is associated with the console window and not with any console buffer. If the `Title()` method fails, it returns `undef`.

Example 9.11 shows a convenient way to change the title when buffers are displayed. The idea behind this is that you might want to change the title every time you display the buffer.

### **Example 9.11 Setting the console window's title**

```
01. use Win32::Console;
02. # Create buffers
03. my $Buf1 = new Win32::Console();
04. my $Buf2 = new Win32::Console();
05. # Hack up the console object and add a "title" key
06. # so we can use it later
07. $Buf1->{title} = "This is Buffer 1";
08. $Buf2->{title} = "This is Buffer 2";
09. # Display buffer 1
10. Show( $Buf1 );
11. $Buf1->Write( "Notice the title of this window.\nHit enter to
continue.\n" );
12. <STDIN>;
13. # Display buffer 2
14. Show( $Buf2 );
15. $Buf2->Write( "Now Notice that the title of this window has
changed!\nHit enter to
continue.\n" );
17.
18. <STDIN>;
20. sub Show
21. {
22.     my( $Buffer ) = @_;
23.     $Buffer->Display();
24.     # Set the console title to reflect which buffer is displayed
25.     $Buffer->Title( $Buffer->{title} );
26. }
```

### **Querying and Setting a Console Window Size**

The size of the console window can be both queried and set by using the `Window()` method:

```
@Window = $Buffer->Window( [ $$PositionFlag, $Left, $Top, $Right,
$Bottom ] );
```

All the parameters are optional, but if one is supplied, all five of them must be supplied.

The parameters for the `Window()` method are defined as follows:

- `$PositionFlag` is the position flag. This flag represents whether the coordinates specified are relative to the current console window or absolute. A value of `0` means the coordinates are relative, and a `1` indicates that the coordinates are absolute.
- `$Left` is the column number of the left side of the window.
- `$Top` is the row number of the top of the window.
- `$Right` is the column number of the right side of the window.
- `$Bottom` is the row number of the bottom of the window.

Passing parameters into `Window()`, the method triggers an attempt to reset the console window size. If the coordinates are within the possible size (refer to the `MaxWindow()` method), the window will be set. If you have multiple buffers, the console window will change only when that buffer is displayed. A buffer must have the `GENERIC_WRITE` access mode set to change the console window size. Refer to the new function.

The `Window()` method returns a `true` (nonzero) value if it successfully sets the window size. If no parameters are passed into `Window()`, an array is returned describing the console window (as illustrated in [Table 9.7](#)). If the method fails, it returns `undef`.

**Table 9.7. The Array Returned from the `Window()` Method**

Array Element	Description
<code>\$Window [0]</code>	The column number of the left side of the rectangle
<code>\$Window [1]</code>	The row number of the top of the rectangle
<code>\$Window [2]</code>	The column number of the right side of the rectangle
<code>\$Window [3]</code>	The row number of the bottom of the rectangle

## Attributes of a Console Buffer

A console buffer is more than just a chunk of memory used to store text. It has specific size and color attributes that can be set and queried, as detailed in the sections that follow.

### Determining and Setting the Console Buffer's Color

The `Attr()` method determines the current color of the console buffer:

```
$Buffer->Attr( [ $$Color ] );
```

The only parameter (`$Color`) is optional and consists of two components: a background color and a foreground color. Any combination of background and foreground variable listed in [Table 9.8](#) (or any combination of variables from [Table 9.9](#)) can be logically OR'ed together to make up the value of this parameter.

The constants in [Table 9.8](#) can be logically OR'ed in any combination. If you want to have bright cyan (blue and green) on a black background, for example, you can specify `BACKGROUND_BLACK | FOREGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY`.

**Table 9.8. Color Constants That Can Be Used with a Console Buffer**

Constant	Description
<code>BACKGROUND_BLUE</code>	The screen color is blue.
<code>BACKGROUND_GREEN</code>	The screen color is green.

**Table 9.8. Color Constants That Can Be Used with a Console Buffer**

Constant	Description
BACKGROUND_RED	The screen color is red.
BACKGROUND_INTENSITY	The screen color is intensified (bright).
FOREGROUND_BLUE	The text color is blue.
FOREGROUND_GREEN	The text color is green.
FOREGROUND_RED	The text color is red.
FOREGROUND_INTENSITY	The text color is intensified (bright).

The variables in [Table 9.9](#) are defined by the `Win32::Console` extension and represent colors based on the constants listed in [Table 9.8](#).

**Table 9.9. Premixed Color Values That Can Be Used with a Console Buffer**

Value	Description
\$BG_BLACK	The screen color is black.
\$BG_BLUE	The screen color is blue.
\$BG_BROWN	The screen color is brown.
\$BG_CYAN	The screen color is cyan.
\$BG_GRAY	The screen color is gray.
\$BG_GREEN	The screen color is green.
\$BG_LIGHTBLUE	The screen color is light blue.
\$BG_LIGHTCYAN	The screen color is light cyan.
\$BG_LIGHTGREEN	The screen color is light green.
\$BG_LIGHTMAGENTA	The screen color is light magenta.
\$BG_LIGHTRED	The screen color is light red.
\$BG_MAGENTA	The screen color is magenta.
\$BG_RED	The screen color is red.
\$BG_WHITE	The screen color is white.
\$BG_YELLOW	The screen color is yellow.
\$FG_BLACK	The text is black.
\$FG_BLUE	The text is blue.
\$FG_BROWN	The text is brown.

**Table 9.9. Premixed Color Values That Can Be Used with a Console Buffer**

Value	Description
\$FG_CYAN	The text is cyan.
\$FG_GRAY	The text is gray.
\$FG_GREEN	The text is green.
\$FG_LIGHTBLUE	The text is light blue.
\$FG_LIGHTCYAN	The text is light cyan.
\$FG_LIGHTGREEN	The text is light green.
\$FG_LIGHTMAGENTA	The text is light magenta.
\$FG_LIGHTRED	The text is light red.
\$FG_MAGENTA	The text is magenta.
\$FG_RED	The text is red.
\$FG_WHITE	The text is white.
\$FG_YELLOW	The text is yellow.

The `Attr()` method returns the current attribute value. If the method sets the attribute, the returning value is the new attribute.

In [Example 9.12](#), the color of the `STDOUT` console buffer is changed to cyan text on a blue background (line 5). Later, the buffer is set back to its original color (line 8). Characters printed after that point are as they were when the script started to run. Notice that the code writes to the buffer using both the `Write()` method (line 6) and `print` (line 9). This is to illustrate how the buffer is modified so that no matter how data is written to the console buffer, it is affected by the buffer's color attribute. The last line (10) is not needed (because the console buffer `$Out` will fall out of scope as the script ends), but it is included to show the proper way to destroy a console buffer.

### **Example 9.12 Modifying the color of a console buffer**

```

01. use Win32::Console;
02. my $Out = new Win32::Console( STD_OUTPUT_HANDLE ) || die;
03. print "We are now going to change the color of the text...\n";
04. my $OrigAttribute = $Out->Attr();
05. $Out->Attr( $FG_CYAN | $BG_BLUE );
06. $Out->Write( "Wow! Check out this color\n" );
07. print "Yes, this color is cool!\n";
08. $Out->Attr( $OrigAttribute );
09. print "Welcome back to the original color!\n";
10. undef $Out;

```

### **Querying and Setting a Console Buffer's Cursor Position**

In addition to setting buffer colors, you can also query and set the current position of the buffer's cursor using the `Cursor()` method:

```
@Cursor = $Buffer->Cursor( [$X [, $Y [, $Size, $Visible] ] );
```

This method takes four optional parameters. Any missing parameter is defaulted to represent the current value (therefore, the attribute that the particular parameter represents will not be modified). The parameters for the `Cursor()` method are defined as follows:

- `$x` and `$y` are optional parameters and represent the X and Y position, respectively, of where the cursor is to be moved to in the buffer. Position `0, 0` is the upper-left corner of the buffer. If these parameters have a value of `-1`, they are ignored—this is used if you want to set the size and visibility of the cursor without setting the cursor's location.
- The `$size` parameter represents the size of the cursor. This value is a bit odd, but blame it on the Win32 API. The value specified is an integer between 1 and 100, which specifies the percentage of the character cell that is filled to indicate the location of the cursor. A typical value is `50` (50% of the character cell is filled). If you do not know what to use for this value, just specify `50`.
- The `$visible` parameter represents whether the cursor is visible. A value of `1` shows the cursor, and `0` hides it.

The returned array reflects the current state of the cursor after any changes have been made by the method.

If successful, the `Cursor()` method returns a four-element array that reflects the current state of the cursor. [Table 9.10](#) describes the values of the array. If the method fails, a `FALSE` value (`undef`) is returned.

**Table 9.10. The Elements of the Array Returned by the `Cursor()` Method**

Array Element	Description
<code>\$Cursor[0]</code>	The X position of the cursor.
<code>\$Cursor[1]</code>	The Y position of the cursor.
<code>\$Cursor[2]</code>	The size of the cursor. This value is between <code>1</code> and <code>100</code> , which represents the percentage of the character cell that is filled to indicate the location of the cursor.
<code>\$Cursor[3]</code>	The visibility of the cursor. (This value is either <code>TRUE</code> or <code>FALSE</code> .)

### **Resizing Console Buffers**

The size of a console buffer can be modified using the `Size()` method:

```
@Size = $Buffer->Size( [$X, $Y] );
```

The two parameters (`$x` and `$y`) are optional.

The first parameter (`$x`) specifies the new number of columns to which the buffer will be set. If this parameter has a value of `-1`, the buffer's number of columns will not change.

The second parameter (`$y`) specifies the new number of rows to which the buffer will be set. If this parameter has a value of `-1`, the buffer's number of rows will not change.

If the two parameters are specified, the buffer's size is altered. The size of the buffer cannot be made smaller than the size of the console window. If either of the parameters passed in have a value of `-1`, that parameter is not altered. If the values `(-1, 255)` are passed in, only the number of rows is changed to 255.

If no values are passed in, the method returns an array consisting of the current number of columns and rows that the buffer has.

If parameters are passed into the `Size()` method, it will return either a `true` or `false` value indicating the success or failure of resizing the buffer. Resizing does not clear the buffer. The only exception to this is when a buffer is made smaller. In this case, data in the columns and rows that are removed is lost.

The code in [Example 9.13](#) illustrates how to both retrieve the size of the buffer and resize it.

### **Example 9.13 Resizing a console buffer**

```
01. use Win32::Console;
02. my $Buffer = new Win32::Console();
03. # Get the current size of the buffer
04. my ($X, $Y) = $Buffer->Size();
05. print "The buffer size is $X columns by $Y rows.\n";
06. # Resize only the number of rows. Give a huge amount!
07. $Buffer->Size( -1, 500 );
08. ($X, $Y) = $Buffer->Size();
09. print "The new buffer size is $X columns by $Y rows.\n";
```

### **Filling Console Buffers with Characters**

A buffer can be filled with a particular character by using the `FillChar()` method:

```
$Buffer->FillChar( $Char, $Number [, $X [, $Y ] ] );
```

The parameters for the `FillChar()` method are defined as follows:

- The `$char` parameter is a character that is to be written to the buffer. This is an actual character such as `A`. If a string with more than one character is specified, only the first one will be used.
- The `$number` parameter specifies how many characters will be written to the buffer.
- The `$x` and `$y` parameters are optional and represent the row and column, respectively, that determine the location in the buffer where the writing of the character specified in the first parameter will begin. Either of these parameters can be specified as `-1` to indicate the

current column (for `$x`) or current row (for `$y`). If either parameter is missing, then `-1` is automatically used for that parameter.

If the `FillChar()` method is successful, it returns the number of characters written to the buffer; otherwise, it returns `undef`. See [Example 9.14](#) for an example of using the `FillChar()` method.

### **Filling Console Buffers with Color**

Just as a buffer can be filled with a character, it can be filled with a color as well. The `FillAttr()` method handles this:

```
$Buffer->FillAttr( $Color, $Number[ , $X [ , $Y ] ] );
```

The parameters for the `FillAttr()` method are defined as follows:

- The `$color` parameter is the color value to be written to the buffer. Any combination of background and foreground color variables, listed previously in [Table 9.9](#) (or any combination of variables from [Table 9.8](#)), can be logically OR'ed together to make up the value of this parameter.
- The `$Number` parameter specifies how many colors will be written to the buffer.
- The `$x` and `$y` parameters represent the row and column, respectively, that determine the location in the buffer where the writing of the character specified in the first parameter will begin. Either of these parameters can be specified as `-1` to indicate the current column (for `$x`) or current row (for `$y`). If either parameter is missing, then `-1` is automatically used for that parameter.

If the `FillAttr()` method is successful, it returns the number of colors written to the buffer; otherwise, it returns `undef`. [Example 9.14](#) illustrates the use of both the `FillChar()` and `FillAttr()` methods.

### **Example 9.14 Filling a console buffer with characters and colors**

```
01. use Win32::Console;
02. my $Buffer = new Win32::Console();
03. $Buffer->FillChar( ".", 1000, 0, 0 );
04. $Buffer->FillAttr( $FG_YELLOW | $BG_BLUE, 1000, 0, 0 );
05. $Buffer->Display();
06. $Buffer->Cursor( 0, 22 );
07. $Buffer->Write( "Hit ENTER to end." );
08. <STDIN>;
```

## **Sound**

One of the exciting things that Win32 platforms can handle is the capability to play back sound files, known as `.WAV` (wave) files. The `.WAV` format is the standard audio format that Win32 understands. Win32 machines typically have several `.WAV` files that come with the operating system.

The `Win32::Sound` extension facilitates the capability to play back these `.WAV` sounds. This can greatly enhance a Perl script by signaling when an event takes place. Unfortunately, this extension does not play back MP3, AU, Windows Media, Real Audio, QuickTime, or other audio formats.

Check with CPAN for the latest audio modules. For Windows Media, Real Audio, and QuickTime audio playback, you might need to interact with their respective audio SDKs using `Win32::OLE` (see [Chapter 5](#), "Automation") or `Win32::API` (discussed later in this chapter).

## Types of Sounds

The following three types of sounds can be played:

- **System beeps.** A system beep is the default beep-like sound that Windows will utter when it does not know what else to do. This is a very simple and quick beep sound. This can be played using the `Play()` function and specifying a system event called `DefaultBeep`.
- **System events.** A system event is a sound associated with a particular event. When Windows starts, for example, it attempts to play the "Start Windows" sound. These sounds are configurable by means of the Sounds Control Panel applet.

The sounds can be configured (and new names added) by means of the Registry. The names and associated paths to wave files are located in the

`HKEY_CURRENT_USER\AppEvents\Schemes\Apps\.Default` Registry key. The current state of the extension supports only the names specified in this Registry key. Other sounds can be configured, but only those under this key are accessible from the extension.

- **Wave Files.** A path to a `.WAV` file can be specified. The path can be either relative to the current directory or a full path. The name of the file can either be a full name such as `Welcome.WAV` or just the filename without the file extension, such as `Welcome`. If no extension is specified, `.WAV` will be assumed.

## Playing Sounds

You can play a sound using the `Play()` function:

```
Win32::Sound::Play( [ $$Sound [, $Flag ] ] );
```

The first optional parameter (`$Sound`) indicates which sound to play. This can be a full or relative path to a `.WAV` file, or it can be the name of a system event.

The second optional parameter (`$Flag`) is a flag consisting of any of the constants listed in [Table 9.11](#). These constants can be logically OR'ed together. If nothing is specified for this parameter, no flags will be used by default.

**Table 9.11. Sound-Playing Flags**

Constant	Description
<code>SND_ASYNC</code>	The sound is played asynchronously. That is, the function returns immediately, and the Perl script continues to process while the sound continues to play. If this

**Table 9.11. Sound-Playing Flags**

Constant	Description
	flag is not specified, the sound plays synchronously, and the function will not return until the sound has finished playing.
SND_LOOP	The sound will play repeatedly until it is purposely stopped. If this flag is specified, <code>SND_ASYNC</code> should also be specified; otherwise, the sound will continue to play, and the Perl script will not return from the play function.
SND_NODEFAULT	Typically, a default beep is played when the specified sound is not able to play (that is, the sound file or the sound name is not found). This flag will prevent this beep from playing.
SND_NOSTOP	When this flag is specified, calling the <code>Play()</code> function while another sound is playing will cause the function to fail without interrupting the current sound.

If no parameters are passed into `Play()`, any currently playing sound will be stopped regardless of the `SND_NOSTOP` flag.

The `Play()` function returns a `TRUE` value if it is successful. The only time this function does not return a `TRUE` value is when it cannot play the specified sound and there is no defined default sound. [Example 9.15](#) shows how to use the `Win32::Sound::Play()` function.

### **Example 9.15 Using the Win32::Sound extension to play a .WAV file**

```
01. use Win32::Sound;
02. my @Events = qw(
03.     MenuPopup
04.     SystemDefault
05.     SystemAsterisk
06.     SystemExclamation
07.     SystemExit
08.     SystemHand
09.     SystemQuestion
10.    SystemStart
11. );
12. foreach my $Sound ( @Events )
13. {
14.     print "Playing $Sound...\n";
15.     Win32::Sound::Play( $Sound, SND_NODEFAULT );
16. }
```

## **Consequences of Playing Sounds**

*Be careful when playing sounds, especially when looping them asynchronously. On Windows 2000 Server, it has been demonstrated that when attempting to break out of a Perl script (with `Ctrl+C`) that is asynchronously looping a sound, the command window freezes and is unable to gracefully terminate.*

## Stopping Sounds from Playing

When a sound is played with the `SND_ASYNC` flag, it will play while the Perl script continues to process. The script can tell the sound to stop playing by calling the `Stop()` function:

```
Win32::Sound::Stop();
```

Any sound that the `Win32::Sound` extension is playing will stop playing. This function always returns a `TRUE` value. The `Stop()` function is demonstrated in [Example 9.16](#). Notice that because the `SND_ASYNC` flag was specified in line 4, the sound plays at while the code between lines 5 and 11 is executed. Line 11 stops the sound from playing.

Calling the `Stop()` function will *not* stop playback of sounds that other processes are playing back.

### **Example 9.16 Stopping a playing sound**

```
01. use Win32::Sound;
02. my $File = "$ENV{SystemRoot}\media\The Microsoft Sound.wav";
03. my $iCount = 4;
04. Win32::Sound::Play( $File, SND_ASYNC );
05. while( -$iCount )
06. {
07.     print "$iCount seconds until the sound stops...\n";
08.     sleep( 1 );
09. }
10. print "Stopping the sound...\n";
11. Win32::Sound::Stop();
12. print "Ahh, the sound has stopped.\n";
```

## Adjusting the Volume

Not only can you use the `Win32::Sound` extension to play sounds, it can also be used to adjust the volume for playback as well. This is done using the `Volume()` function:

```
Win32::Sound::Volume( [ $$Left [, $Right ] ] );
```

The optional first parameter indicates the left channel volume level. This can be any numeric value from 0 to 65535, where 0 is equivalent to mute and 65535 is the highest volume. The value can also be specified as a string representing a percentage of the possible volume level (0 percent through 100 percent).

The optional second parameter indicates the right channel volume level. This can be any numeric value from 0 to 65535, where 0 is equivalent to mute and 65535 is the highest volume. The value can also be specified as a string representing a percentage of the possible volume level (0 percent through 100 percent).

If only one parameter is specified, it is used for both left and right channels. The volume setting only applies to the default sound device.

If no parameters are passed in, the function returns a value indicating the current volume settings. If the return value is in an array context, the first element of the returning array is the left volume setting, and the second array element is the right volume setting. These are numeric values from 0 to 65535. If the return context is a nonarray context, the return value is a packed value. To unpack it, use the following:

```
( $Right, $Left ) = unpack( "S2", Win32::Sound::Volume() );
```

If parameters were passed in and the function was successful, it returns `TRUE (1)`; otherwise, it returns `FALSE (0)`. If no parameters were passed in, either an array of two integer values is returned or a single packed numeric value is returned for a return array context and return scalar context, respectively.

[Example 9.17](#) shows how easy it is to modify the volume. First, the code plays a looping, asynchronous sound (line 3), then it decreases the volume from 100 percent to 0 percent (lines 7 through 16), then it increases the volume from 0 percent to 100 percent (lines 18 through 28). One of the interesting lines to pay attention to is line 12, where the volume is set by passing in two percentage strings (one for each channel—left and right). Line 13 fetches the new left and right channel volume level. This is a value between 0 and 65535.

Line 23 sets the volume level by specifying a percentage string. This is just like line 12 except that we pass in only one percentage, which is used for both left and right channels. Line 12 passes in both channels independently. Lines 24 and 25 use the alternative way of discovering the current volume level. Notice that we have to unpack the channel levels value returned by the `Volume()` function. Line 24 retrieves the packed value in a scalar (nonarray) context. Line 25 unpacks the values as two unsigned shorts (16-bit values).

## The Volume Bug

*The `Win32::Sound::Volume()` function suffers from a pretty major bug. The volume levels are not processed correctly. When you pass in any volume level (to set the volume), it will silence the right channel and apply a quieter value than what you specified. This occurs regardless of whether you specify both channels or one and regardless of whether you specify a percentage instead of a specific numeric value.*

*The bug also is present when you fetch the volume level as an array. However, if you fetch the volume level as a packed scalar value, the bug does not affect the outcome. In other words, as of version .46, the only reliable way to discover the current volume level is to call the following:*

```
$PackedVolume = Win32::Sound::Volume();
($Right, $Left) = unpack( "S2", $PackedVolume );
```

*Until the bug has been fixed, there is no reliable way to modify the volume level with this*

*extension.*

### **Example 9.17 Adjusting the volume**

```
01. use Win32::Sound;
02. my $Volume = 100;
03. Win32::Sound::Play( "SystemExclamation", SND_ASYNC | 
SND_LOOP );
04. print "Pausing for a few seconds so that we can hear the
sound...\n";
05. sleep( 3 );
06. print "Decreasing volume...\n";
07. while( $Volume- )
08. {
09.     my $Percentage = "$Volume%";
10.    my( $Right, $Left );
11.    print "\r\t$Percentage";
12.    Win32::Sound::Volume( $Percentage, $Percentage );
13.    ( $Right, $Left ) = Win32::Sound::Volume();
14.    print " (left: $Left, right: $Right)", " " x 20;;
15.    Win32::Sleep( 25 );
16. }
17. print "\rIncreasing volume...", " " x 20,, "\n";
18. while( $Volume++ <= 100 )
19. {
20.     my $Percentage = "$Volume%";
21.     my( $Right, $Left );
22.     print "\r\t$Percentage";
23.     Win32::Sound::Volume( $Percentage );
24.     $CurrentVolume = Win32::Sound::Volume();
25.     ( $Right, $Left ) = unpack( "S2", $CurrentVolume );
26.     print " (left: $Left, right: $Right)", " " x 20;
27.     Win32::Sleep( 25 );
28. }
29. print "\rStopping the sound...", " " x 20,, "\n";
30. Win32::Sound::Stop();
31. print "Ahh, the sound has stopped.\n";
```

## **Win32 API**

Almost all of the Win32-specific extensions make use of some Win32 API function calls. Many of these extensions exist only because Perl does not have any native way of calling these functions. This is where the `Win32::API` extension comes into play.

This extension provides a facility to make calls into the Win32 API. Specifically, the extension enables a Perl script to make any call into any dynamic link library (DLL). These DLLs are not limited to only Win32-specific libraries; any DLL can be used.

### **Warning**

A word of warning regarding the `Win32::API` extension: This extension gives full, unlimited access to any DLL that a user can find. This can easily cause damage to your computer by corrupting data files, the Registry, and hard drives; other damage is possible. Before experimenting with DLL functions, you should consult the library's documentation. Before doing something with this library, you should think very hard about the results it can lead to.

This section covers the use of the `Win32::API` extension. A description of how DLLs work, what functions are available from various DLLs, and other topics such as defining a void pointer, string pointer, or `NULL` pointer goes far beyond the scope of this book. You can find many sources for information, such as the Microsoft Developer Network (MSDN), C or C++ compiler's documentation, any good C Win32 programming book, and the Internet.

## Calling Functions from a DLL

When you want to call a function from a DLL, you need to load the DLL and import the function. You can do this by creating a new API object:

```
$Api = new Win32::API( $Dll, $FunctionName, $InputPrototype,  
$ReturnType );
```

The parameters for this function call are as follows:

- The `$Dll` parameter is the name of the DLL file. This can be either a relative filename or a full path. The name of the file does not need to include the `.DLL` extension. If no path is specified, the system will look for the file in the current directory and the Windows and Windows system directories, and then the path will be checked for the file.
- The `$FunctionName` parameter is the name of the function to be loaded and used. This is a case-sensitive parameter.
- The `$InputPrototype` parameter is an input prototype for the function. This value is an anonymous array or a string. The elements of the array or string can consist of any of the constant values in [Table 9.12](#). String support was introduced with version .20.
- The `$ReturnType` parameter is the output value prototype. This value is a single constant listed in [Table 9.12](#).

**Table 9.12. Data Type Constants**

Constant	Description
I	A short number such as an <i>int</i> or <i>short</i> (16-bit) C data type.
N	A long number. This is the equivalent to a <i>long</i> (32-bit) C data type.
L	A long number. This is the equivalent to a <i>long</i> (32-bit) C data type. This specific data type is typically not used. Instead, use N.
P	A pointer. This can be a pointer to a character string, a Unicode string, a void pointer, a structure pointer, and so on.
F	A floating point number. This is equivalent to a <i>float</i> C data type. This was introduced in

**Table 9.12. Data Type Constants**

Constant	Description
version .20.	
D	A double number. This is equivalent to a <i>double</i> C data type. This was introduced in version .20.

Take, for example, the `GetTickCount()` function in the `kernel32.dll` library, which takes no parameters but returns a `DWORD`. Its C prototype looks like this:

```
DWORD GetTickCount(VOID);
```

Because `GetTickCount()` has no input parameters (it is `void`), the third parameter will be an empty anonymous array: `[]`. The function does return a `DWORD`, which is a 32-bit number, the same as `long`; therefore, the fourth parameter will be an `N` (representing a `long` number). To create a `Win32::API` object for this function, use the following:

```
$Api=new Win32::API("kernel32.dll", "GetTickCount", [], N);
```

Another example is the `kernel32.dll`'s `GetSystemDirectory()` function. Its C prototype is as follows:

```
UINT GetSystemDirectory( LPTSTR pszBuffer, UINT iSizeOfBuffer );
```

To construct the third parameter for the `new` command, you need to analyze the input parameters to this function. The first input parameter is a pointer to a string, so you will use a `P` value. It is followed by an unsigned integer (a 16-bit `int`), so an `I` data type will suffice. The resulting anonymous array that will be used will be `[ P, I ]`. The return value of the function is a `UINT`, which is represented (as already seen) by a short (16-bit integer); therefore, you will specify an `I` value.

This means that the API object will be created using the following:

```
$ApiObject = new Win32::API("kernel32", "GetSystemDirectory",
[ P,, I ], I );
```

The result of this function will be either a new `Win32::API` object or `undef`. After the `Win32::API` object has been created, you can actually call the function by using the `Call()` method:

```
$ApiObject->Call( $Input1, $Input2, $Input3 ... );
```

The input parameters are defined by the function prototype. The method returns the result from the function.

## Using Win32::API

Now that the mechanics have been discussed, we should walk through a few examples for this to make any sense. You will have to use several tricks to make use of the functions. Some of these might not be obvious.

In the example cited earlier that called the `GetSystemDirectory()` Win32 function, the result was a Win32 API object. Before the `Call()` method can be executed, there must be some preparations first. The API function expects the first pointer to point to a character string, and the second parameter (the `UINT`) is the number of bytes that have been allocated for the pointer. Because the pointer must point to allocated memory, memory must first be allocated. This is accomplished by creating a string of characters with the following:

```
$String = "\x00" x 1024;
```

This will allocate a string of 1,024 bytes. It is a good idea to fill the string with `NULL` characters `\x00` because any API function that expects an incoming string will expect it to be terminated with a `NULL` character.

After a 1KB string has been allocated, the `Call()` method can be called:

```
$Result = $ApiObject->Call( $String, length( $String ) );
```

Notice that the string that was allocated is passed in as the pointer value (as specified by the `P` constant), and the second value is the size of the allocated buffer (which will be interpreted as an integer value because the `I` constant defined it). The result was defined as a long data type (the `N` constant). When the method call returns, the returned value is converted into a normal Perl-based integer value. In this case, it represents how many characters were stored into the allocated buffer.

The resulting path stored in the buffer will be a proper C-based `NULL` terminated string, so there is a need to extract everything up to the first `NULL` character:

```
($Dir) = ( $String =~ /(.*?)\x00/ );
```

That is it! [Example 9.18](#) shows the code in its entirety.

### Example 9.18 Using Win32::API to retrieve the system directory

```
01. use Win32::API;
02. my $ApiObject = new Win32::API( "kernel32",
03.   "GetSystemDirectory", [ P, I ], I );
04. my $String = "\x00" x 1024;
05. $ApiObject->Call( $String, length( $String ) );
06. my($Dir) = ( $String =~ /(.*?)\x00/ );
07. print "The system directory is '$Dir'\n";
```

## Specifying a NULL Pointer

When you want to specify a `NULL` pointer when calling the `Call()` method, just pass in a `0` value for the pointer argument. For example:

[View full width]

```
my $Api = new Win32::API( 'my.dll', 'MyFunction', [P], I )
||
die;
$Result = $Api->Call( 0 );
```

By passing in a `0` value for the pointer, `Win32::API` will interpret this to be the same as passing in a `NULL` pointer.

## Using Structures with `Win32::API`

To demonstrate another, more complex example, consider the `GetVersionEx()` Win32 API function. This returns information regarding the Windows version numbers. This example is interesting because it deals with a data structure.

The `GetVersionEx()` prototype function is as follows:

```
BOOL GetVersionEx( OSVERSIONINFO *pOSVersion );
```

The return type `BOOL` can be held in an `int`, so the constant `I` will be used. The input parameter is a pointer to a data structure (`P`). Hence, the input parameter anonymous array will be `[ P ]`.

Considering this, the API object will be created using this:

```
$ApiObject = new Win32::API( "kernel32", "GetVersionEx", [P], I );
```

To continue, you need to allocate memory to hold the data structure. The structure is prototyped as follows:

```
typedef struct _OSVERSIONINFO
{
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[ 128 ];
} OSVERSIONINFO;
```

Because a `DWORD` is equivalent to a 32-bit `long` data type, you will use the `long` instead. The `TCHAR` data type is either a character or a Unicode character, depending on how the extension was compiled. If the extension was compiled enabling Unicode, the 128-element `TCHAR` array will be

256 bytes long (because a Unicode character is 16 bits wide). Before allocating the memory, you need to determine whether you are using Unicode. This can be done using the `IsUnicode()` function:

```
Win32::API::IsUnicode();
```

The function returns a `TRUE` value if the extension was compiled to always use Unicode; otherwise, it returns `FALSE`.

Therefore, to create the data structure, you could use the following:

```
# Determine how large a character is
$CharSize = 1 + Win32::API::IsUnicode();
$Struct = pack( "L5c*", ( (0) x 5 ), ( (0) x ( $CharSize *
128 ) ) );
$StructSize = length( $Struct );
$Struct = pack( "L5c*", $StructSize, ( (0) x 4 ), ( (0) x
( $CharSize * 128 ) )
);
```

This is a bit of a hack, but the memory structure requires that the first `DWORD` (`dwOSversionInfoSize`) be set to have the size of the memory structure. So by packing it once, you can then get the length of the structure so you can repack it with this value.

Now that the structure has been created, submit it into the API function:

```
$Result = $ApiObject->Call( $Struct );
```

This particular function returns a `0` if it failed; if it successfully filled out the structure, it returns a nonzero value. If the function returned a nonzero value, you need to unpack the structure into variables:

```
( $StructSize, $Ver{major}, $Ver{minor}, $Ver{build},
$Ver{platform},
$Ver{servicepack} ) = unpack( "L5a*", $Struct );
```

The last thing that must be done is to rip out any `NULL` characters that are left in the `$Ver{servicepack}` variable:

```
$Ver{servicepack} =~ s/\x00//g;
```

To see the code in its entirety, see [Example 9.19](#). Notice that line 5 determines how large the string should be (in bytes), depending on whether the `Win32::API` extension was compiled for Unicode or not. In this case, we assume that a character is 1 byte in size. Then we add one more if Unicode is used.

Line 6 constructs an empty packed structure of 5 long (each 32 bits in size) data types followed by 128 characters. That will end up being 128 or 256 bytes of characters, depending on whether or not Unicode is involved. Interestingly enough, the structure was packed with zeros; in other words, it is an empty structure. We created it only so that we can determine how large it is. Next, line 7 computes how long the structure is (in bytes), which is used when we repack the structure in line 8, this time including its data.

The reason for this little dance is because the first element in the structure we are creating needs to be set with the physical size of the structure (the Win32 `GetVersionEx()` function expects this). This is why we had to first build the structure to discover its size. Then we rebuild the structure in line 8; the first long parameter is set to the size of the structure and then the remaining longs and character buffer are set to `0`. These will be filled out by the `GetVersionEx()` function.

Line 11 calls the operating system's `GetVersionEx()` function, passing in the packed structure the script has created. If the function is successful, line 13 unpacks the structure. This results in the `@Ver` array consisting of the information that `GetVersionEx()` set in the packed structure. Pay close attention to line 14. This is where the script cleans up the fifth element of `@ver`. This element is the character string that was passed back. It is important to clean up the string because there might be trailing `\0` characters. Perl will attempt to print those characters unless they are removed.

### **Example 9.19 Retrieving the Windows version information using `Win32::API`**

```
01. use Win32::API;
02. my $ApiObject = new Win32::API( "kernel32", "GetVersionEx",
[ P ], I );
03. # Find how many bytes we need for the char string-based on
whether
04. # we are dealing with Unicode or not
05. my $CharSize = 1 + Win32::API::IsUnicode();
06. my $Struct = pack( "L5c*", ( (0) x 5 ), ( (0) x ( $CharSize * 128 ) ) );
07. my $StructSize = length( $Struct );
08. $Struct = pack( "L5c*", $StructSize,
09.                  ( (0) x 4 ),
10.                  ( (0) x ( $CharSize * 128 ) ) );
11. if( my $Result = $ApiObject->Call( $Struct ) )
12. {
13.     my @Ver = unpack( "L5a*", $Struct );
14.     $Ver[5] =~ s/\x00//g;
15.     print "This is Windows $Ver[1].$Ver[2] " .
16.           "build $Ver[3] $$Ver[5]\n";
17. }
18. else
19. {
20.     print "The call to GetVersionEx() failed.\n";
21. }
```

Just to better illustrate how `Win32::API` can be used, look at [Example 9.20](#). This is a simple script that accepts an Internet URL and looks up any HTTP cookies associated with it. This is the same way that Web-based applications (such as Internet Explorer) determine what cookies to send to a Web server.

### **Example 9.20 Examining Internet cookies**

[View full width]

```
01. use Win32::API;
02. my $ApiObject = new Win32::API( "wininet", "InternetGetCookie",
[ P, P, P, P ], I ) || die;
03. my $Cookie = "\x00" x 1024;;
04. my $Url = shift @ARGV || die;
05. my $pdwCookieLength = pack( "L", length( $Cookie ) );
06.
07. if( my $Result = $ApiObject->Call( $Url, undef, $Cookie,
$pdwCookieLength ) )
08. {
09.     $Cookie =~ s/\s+$///;
10.    $Cookie =~ s/\c@+$//;
11.    print "Cookie: '$Cookie'\n";
12. }
13. else
14. {
15.    print "Could not find a cookie.\n";
16. }
```

## **Common Routines**

Because many Win32 functions require allocating memory blocks of a specific size and handling Unicode strings, it is handy to have some functions around that facilitate such needs. [Example 9.21](#) lists some of these routines.

These functions include the following:

- `Alloc( $Size )`. This function returns an allocated string of the specified size. The string is filled with `NULL` characters.
- `AnsiToUnicode( $String )`. This function returns a Unicode version of the passed-in ANSI string.
- `UnicodeToAnsi( $String )`. This function returns an ANSI version of the passed-in Unicode string.

### **Example 9.21 Commonly used routines**

```
01. sub Alloc
02. {
03.     my( $Size ) = @_;
04.     my( $Pointer ) = "\x00" x $Size;
05.     return $Pointer;
06. }
07.
08. sub AnsiToUnicode
09. {
10.     my( $Unicode ) = @_;
11.     $Unicode =~ s/(.)/$1\x00/gis;
12.     return $Unicode;
```

```

13. }
14.
15. sub UnicodeToAnsi
16. {
17.     my( $Ansi ) = @_;
18.     $Ansi =~ s/(.)\x00/$1/gis;
19.     return $Ansi;
20. }

```

## Impersonating a User

Sometimes a script needs to log on as a different user—what is known as impersonation. Consider a Web CGI script that enables a user to modify her user account information (such as her full name) or to make modifications to her home directory. The script could accept the user's userid and password as input parameters and attempt to log on as that user. If successful, the script could have access to files and directories to which only she has permission. Services use this feature quite often, such as Web servers and Telnet daemons.

The `Win32::AdminMisc` extension provides an impersonation function called `LogonAsUser()`:

```
Win32::AdminMisc::LogonAsUser( $Domain, $User, $Password[ ,  
$LogonType ] );
```

The parameters for the `LogonAsUser()` function are defined as follows:

- `$Domain` is the name of the domain where the user account exists. If this parameter is an empty string (""), the current domain is assumed.
- `$User` is the name of the user account to be impersonated.
- `$Password` is the password for the user account specified in the second parameter.
- `$LogonType` is optional and represents the type of logon to be attempted. This can be any value from [Table 9.13](#).

**Table 9.13. The Logon Types Used with `Win32::AdminMisc::LogonAsUser()`**

Logon Type	Description
<code>LOGON32_LOGON_INTERACTIVE</code>	This type caches logon information.
<code>LOGON32_LOGON_BATCH</code>	This logon type is similar to the <code>LOGON32_LOGON_NETWORK</code> type, except that it can be used in conjunction with <code>CreateProcessAsUser()</code> .
<code>LOGON32_LOGON_SERVICE</code>	This is the logon type used by services. For this type to be used, the impersonated account must have the privilege to log on as a service.
<code>LOGON32_LOGON_NETWORK</code>	This logon type is designed for quick authentication. It is quite limited because the <code>CreateProcessAsUser()</code> function is not

**Table 9.13. The Logon Types Used with `Win32::AdminMisc::LogonAsUser()`**

Logon Type	Description
	supported using this impersonation logon type. This logon type is not recommended.

The `LogonAsUser()` function requires the script calling the function to have the following privileges:

- `SE_TCB_NAME`. Act as part of the operating system.
- `SE_CHANGE_NOTIFY_NAME`. Bypass traverse checking. (This privilege needs to be granted unless the account that the script runs under is either the local system or a member of the Administrators group.)
- `SE_ASSIGNPRIMARYTOKEN_NAME`. Replace a process level token.

### Tip

A bug that crept into the `Win32::AdminMisc` extension causes `LogonAsUser()` to incorrectly return a `1` even if the logon was unsuccessful. A simple way to verify that the impersonation was successful is to check the logon name immediately after the call to `LogonAsUser()`:

```
Win32::AdminMisc::LogonAsUser( $Domain, $User, $Password );
$UserName = Win32::AdminMisc::GetLogonName();
print "Failed to logon" if( lc $User ne lc $UserName );
```

After you have successfully logged on as another user, you might need to check who you are logged on as. `Win32::AdminMisc::GetLogonName()` does just this. Unlike the `Win32::LoginName()` function, `GetLogonName()` correctly reports the logon name of an impersonated user:

```
Win32::AdminMisc::GetLogonName();
```

The function takes no parameters and returns the name of the current user. If the logged-on user is impersonated, this function returns the impersonated username.

After a script has impersonated a user, it will eventually want to stop impersonation. To stop impersonating another user, you use `LogoffAsUser()`:

```
Win32::AdminMisc::LogoffAsUser();
```

This function always returns a `TRUE` value. [Example 9.22](#) demonstrates how to impersonate a user.

### **Example 9.22 Impersonating a user**

```
01. use Win32::AdminMisc;
02. my( $Domain, $User, $Password ) = @ARGV;
```

```

03. Win32::AdminMisc::LogonAsUser( $Domain, $User, $Password );
04. $Name = Win32::AdminMisc::GetLogonName();
05. if( "\L$User" ne "\L$Name" )
06. {
07.     print "The logon failed.\n";
08. }else{
09.     print "Successfully logged on as $Name\n";
10.    Win32::AdminMisc::LogoffAsUser();
11. }

```

## Miscellaneous Win32 Functions

Many miscellaneous functions come with the standard `Win32` extension. These functions include some important and quite-often-overlooked necessities.

Even though all these functions are accessed through the `Win32` namespace, such as

```
Win32::DomainName();
```

not all of them are part of the `Win32` extension. Some of these functions (such as the `DomainName()` function) are built into Win32 Perl directly. Therefore, you can call these functions as you would a normal built-in Perl function: without loading any extension. Regardless of whether you are calling a built-in function or calling a Win32 extension-specific function, you must still specify the Win32 namespace.

## Built-in Win32 Functions

Win32 Perl has built-in functions that are only applicable to Win32 platforms; they won't work on other platforms. These functions do not require loading any extensions. Even though there is no need to load any extensions, these functions belong to the Win32 namespace. Therefore, to call any of them, you need to prefix them with `Win32::`.

Even though there is no need to load the `Win32` extension for these functions, it is good practice to do so. Besides, it does not do any harm to explicitly load the extension. To top it off, you should probably load the extension anyhow. Way back before version 5.005, there were more built-in functions that have since been moved into the `Win32` extension proper. Any script that failed to load the `Win32` extension would break when it was run under version 5.005 or higher.

All the examples in this section will explicitly load the `Win32` extension, even though it is not required.

### ***Discovering Build and Version Numbers***

The version (or to be more precise, the build number) of Perl can be obtained with a call to `PerlVersion()`:

```
Win32::PerlVersion();
```

The build number is returned by a call to this function. A call to this function using ActiveState's Win32 Perl 5.003\_07 Build 316 results in the returned text string `Build 316`, for example:

This is similar to the `$]` variable.

## Note

The `PerlVersion()` function has been deprecated in Perl 5.005 and no longer exists. Scripts that make a call into `Win32::PerlVersion()` using Perl 5.005 or higher will fail.

## Retrieving OS Version Information

You can retrieve information regarding the operating system's version by using the `GetOSVersion()` function:

```
@Version = Win32::GetOSVersion();
```

An array is returned consisting of the values listed in [Table 9.14](#).

**Table 9.14. Array Elements Returned from `Win32::GetOSVersion()`**

Element	Description
<code>\$Version[0]</code>	The Win32 Service Pack number. This is also known as the Corrective Service Disk (CSD) number. This is a string value similar to "Service Pack 3". If no service packs have been applied, this value is an empty string.
<code>\$Version[1]</code>	Major version number. If you are running NT 4.0, this value would be <code>4</code> .
<code>\$Version[2]</code>	Minor version number. If you are running NT 4.0 or 3.51, this value would be either <code>0</code> or <code>51</code> , respectively.
<code>\$Version[3]</code>	Build number. This is the Win32 build number.
<code>\$Version[4]</code>	Platform ID. This value describes the type of Win32 operating system running on the machine. Possible values: <ul style="list-style-type: none"><li><code>0</code> = Win32 (Win16)</li><li><code>1</code> = Win32 (Win95/98/ME)</li><li><code>2</code> = Win32 (Win NT/2000/XP)</li></ul>

## Retrieving Win32 Machine Type Information

Two functions describe the type of Win32 machine that the script is running on, be it Windows NT or Windows 95:

```
Win32::IsWinNT();
Win32::IsWin95();
```

Both functions return `TRUE` if their respective type of machine is represented; otherwise, `0` is returned. [Example 9.23](#) demonstrates how to use the `IsWinNT()` and `IsWin95()` functions.

### **Example 9.23 Using the `Win32::IsWinXX()` functions**

```
01. print "This is a Windows NT/2000/XP machine.\n";
if( Win32::IsWinNT() );
02. print "This is a Windows 95/98/ME machine.\n";
if( Win32::IsWin95() );
```

## ***Discovering Various Names***

The `Win32` extension provides functions to discover the name of the current user, the computer name, and the current domain name:

```
Win32::LoginName();
Win32::NodeName();
Win32::DomainName();
```

These will return the user's login name, the computer's network (node) name, and the user's logon domain name, respectively, or the string "`<Unknown>`" if the name could not be determined for some reason. [Example 9.24](#) shows the use of these functions.

### **Example 9.24 Discovering various names**

```
01. my $User = Win32::LoginName();
02. my $Machine = Win32::NodeName();
03. my $Domain = Win32::DomainName();
04. print "Your user name is $User\n";
05. print "Your machine name is $Machine\n";
06. print "Your domain name is $Domain\n";
```

## ***Discovering the Current Working Directory***

Every executable that runs on a Win32 machine can have a current working directory. (It is also possible that an executable can have no working directory, but I digress.) Perl is no exception. If there is a need to discover what the current working directory is, use the `GetCwd()` function:

```
Win32::GetCwd();
```

This returns a text string that represents the current working directory. This will be in a format similar to `c:\\Program Files\\MyDirectory`. See [Example 9.25](#) for a demonstration of how to properly use the `GetCwd()` function.

## ***Changing the Current Working Directory***

Just as you can retrieve the current directory, you can also change it using the undocumented `SetCwd()` function:

```
Win32::SetCwd( $Path );
```

The only parameter it takes is a string representing the path that is to become the current working directory. This path can be a relative or full path, but it cannot be a UNC.

If the function is successful, it returns a `true (1)` value, and the working directory is set to the new path; otherwise, the function fails and returns `undef`. See [Example 9.25](#) and [Example 9.26](#) for a demonstration of how to properly use the `SetCwd()` function.

### ***Discovering the Next Available Drive***

Sometimes it is important to learn what the next available drive letter is. This represents a drive that is not mapped to a network share and has no CD-ROM, floppy disk drive, or hard drive attached to it. By determining the next available drive, you can figure out which drives do exist on your system. This information can be obtained by the `GetNextAvailDrive()` function:

```
Win32::GetNextAvailDrive();
```

This function returns the first free drive letter not in use. The returned value is a string that represents the drive's root path, such as `D:\` and `M:\`.

The most serious drawback of this function is that it looks at each drive starting with `A:\` and working to `Z:\`. It reports back the first drive letter without any attached device. Because many machines have only one floppy disk drive, this means the first available drive is `B:\`; therefore, this function will typically return `B:\` on most machines. This issue is addressed in Perl version 5.005 and higher, in which the function examines only drives `C:\` through `Z:\`.

The `Win32::AdminMisc` extension has a series of similar functions that not only report the drive letters in use but also what type of device the drive letter represents (a CD-ROM drive, floppy disk drive, hard drive, network share). Refer to [Chapter 3](#), "Administration of Machines," for more information.

### ***Discovering the File System of a Drive***

Different drives can be formatted using different file systems. A Windows NT machine might have one drive formatted as FAT, for example, another as NTFS, and yet another as HPFS (although NT's support for HPFS has been dropped). To help determine the format, there is the `FsType()` function:

```
Win32::FsType();
```

This function takes no parameters and will return a text string representing the file system (such as `FAT`, `NTFS`, or `HPFS`). There is a catch here, however. *The drive that is reported is the current drive based on the current working directory*. To check different drives, you must change the current working directory first (refer to the section "[Discovering the Current Working Directory](#)" earlier in

this chapter). Additionally, because UNC paths cannot be set as current working directories (by using the Win32 extension), a script cannot discover a file system type for a UNC.

In Perl 5.005 and higher, this function has been changed to return either a text string or an array, depending on the return value's context. The array consists of the elements in [Table 9.15](#).

**Table 9.15. Array Elements Returned From Perl 5.005's Win32::FsType() Function**

Array Element	Description
Element 0	String representing the text of the file system type (such as FAT or NTFS).
Element 1	A numeric value representing different flags that the volume (or drive) has set. See <a href="#">Table 9.16</a> .
Element 2	A numeric value representing the maximum number of characters that can be used in a file or directory name.

**Table 9.16. Possible Volume Flags**

Constant	Value	Description
FS_CASE_IS_PRESERVED	0x00000002	The file system preserves the case of filenames when it places names on disk.
FS_CASE_SENSITIVE	0x00000001	The file system supports case-sensitive filenames.
FS_UNICODE_STORED_ON_DISK	0x00000004	The file system supports Unicode in filenames as they appear on disk.
FS_PERSISTENT_ACLS	0x00000008	The file system preserves and enforces ACLs. For example, NTFS preserves and enforces ACLs; FAT does not.
FS_FILE_COMPRESSION	0x00000010	The file system supports file-based compression.
FS_VOL_IS_COMPRESSED	0x00008000	The specified volume is a compressed volume; for example, a DoubleSpace volume.
FILE_SUPPORTS_ENCRYPTION	0x00020000	The file system supports the Encrypted File System (EFS).
FILE_SUPPORTS_OBJECT_IDS	0x00010000	The file system supports object identifiers.
FILE_SUPPORTS_REPARSE_POINTS	0x00000080	The file system supports reparse points.
FILE_SUPPORTS_SPARSE_FILES	0x00000040	The file system supports sparse files.
FILE_VOLUME_QUOTAS	0x00000020	The file system supports disk quotas.
FILE_SUPPORTS_REMOTE_STORAGE	0x00000100	The file system supports remote storage.

[Example 9.25](#) shows a typical use of the `Win32::FsType()` function. Notice that line 5 attempts to change the current directory using an undocumented Win32 extension function (`SetCwd()`), which was covered in the earlier section "[Discovering the Current Working Directory](#)."

### **Example 9.25 Discovering a drive's file system type**

```
01. PrintDriveType( "C:" );
02. PrintDriveType( "D:" );
03.
04. sub PrintDriveType
05. {
06.     my( $NewDrive ) = @_;
07.     my( $Drive ) = ( Win32::GetCwd() =~ /(^(\S:))/ );
08.     print "The file system for $NewDrive is: ";
09.     if( Win32::SetCwd( $NewDrive ) )
10.    {
11.        print Win32::FsType() . "\n";
12.        Win32::SetCwd( $Drive );
13.    }
14.    else
15.    {
16.        print "Unable to determine ($^E)\n";
17.    }
18. }
```

#### **Tip**

The `Win32::AdminMisc::GetVolumeInfo()` function effectively does the same thing as the Perl 5.005 version of `Win32::FsType()`. The only real difference is that with the `GetVolumeInfo()` function, any drive (or UNC) can be specified.

[Example 9.26](#) is similar to [Example 9.25](#) except that it takes advantage of the additional information returned by the function when using Perl 5.005 and higher.

### **Example 9.26 Discovering a drive's file system type with Perl 5.005 and-higher**

```
01. PrintDriveType( "C:" );
02. PrintDriveType( "D:" );
03.
04. sub PrintDriveType
05. {
06.     my( $NewDrive ) = @_;
07.     my( $Drive ) = ( Win32::GetCwd() =~ /(^(\S:))/ );
08.     print "Drive $NewDrive\n";
09.     if( Win32::SetCwd( $NewDrive ) )
10.    {
11.        my( @Volume ) = Win32::FsType();
12.        my( $fEncSupport ) = $Volume[2] && 0x00020000;
13.        my( $fQuotaSupport ) = $Volume[2] && 0x00000020;
14.        print "\tFile system is $Volume[0].\n";
```

```

15.     print "\tLargest file name can be $Volume[2] characters
long..\n";
16.     print " \t" . ( $fEncSupport)? "Does":"Does not" ) . "
support encryption\n";
17.     print " \t" . ( $fQuotaSupport)? "Does":"Does not" ) . "
support quotas\n";
18. }
19. else
20. {
21.     print "\tUnable to locate drive ($^E).\n";
22. }
23. }

```

### ***Obtaining the Last Win32 Error***

Normally, when Win32 runs into an error, it updates a global error state that a program can query. This is useful because a call into some Win32 function can result in an error for many reasons. Your script can query Win32 for the last error generated using `GetLastError()`:

```
Win32::GetLastError();
```

This will return a numeric value indicating the Win32 error condition. Generally, a return value of `0` indicates that there is no error.

Use this function with caution because it can be very misleading. Many extensions will call several Win32 API functions before returning to your script. Even if a call into some extension fails, the Win32 error state might be reset by the extension. For example, consider a function that copies a file. If there was a failure in copying, the Win32 error state might be set to the error. However, if the extension tries to clean up by deleting the partially copied file, then the Win32 error state has been reset to "no error." This is because the last function was successful, hence `Win32::GetLastError()` returns a value of `0` ("the command completed successfully"). This is why most extensions have their own function to report any error state.

Calling this function is the equivalent of querying the `$^E` variable. In a numeric context, it returns the same thing that `Win32::GetLastError()` would return. In a string context, it returns the textual representation of the Win32 error. However, `$^E` is not guaranteed to always be set. Treat this variable as you would a call to `Win32::GetLastError()`.

### ***Setting the Last Error State***

There are times when you might want to convince the Win32 operating system that a particular error has occurred. For example, if a function reacts differently when there is an error state, you could explicitly set the error state to get that functionality. Just in case you want to tell Win32 that it should believe that a particular error state exists, you can set the last Win32 error. This is done by calling the `Win32::SetLastError()` function.

```
Win32::SetLastError( $Error );
```

The only parameter is a numeric value that represents a valid Win32 error state.

Most users will never bother calling this function. However, if you do need to convince Win32 that an error exists, this is how you can do it.

### ***Making Sense of Win32 Errors***

Most Win32 errors make no sense to us humans. They are just numeric values such as 5 or 123. The `Win32::FormatMessage()` function can actually convert these error values into a nice, human-readable text string.

```
Win32::FormatMessage( $Error );
```

The only parameter it takes is a numeric value that represents a valid Win32 |error state.

The function returns a text string that indicates the meaning of the error string. However, there are many errors that this function cannot decipher, such as if a network protocol generated an error. In this case, Win32 does not know what the protocol error means.

[Example 9.27](#) accepts any number of Win32 error numbers and prints out their descriptions.

### ***Example 9.27 Discovering the Win32 error state***

```
01. foreach my $ErrorNum ( @ARGV )
02. {
03.   print "Error $ErrorNum is: ",
Win32::FormatMessage($ErrorNum), "\n";
04. }
```

### ***Obtaining the System Tick Count***

Not everyone uses the system tick count, but for those who do, they can obtain it easily. The system tick count represents the number of milliseconds that have passed since the last time the operating system was booted.

```
Win32::GetTickCount();
```

Because the tick count is represented by a DWORD (a 4-byte unsigned long), the largest value is 4,294,967,295. This equates to about 49.7 days. In other words, the tick count value will reset to 0 every 49.7 days. When it does reset, it will start counting up again.

[Example 9.28](#) demonstrates using the `Win32::GetTickCount()` function. Notice that the tick count increases several times per second.

### ***Example 9.28 Querying the system tick count***

```
01. my $Count = 200;
02. while( --$Count )
03. {
```

```
04. print "Time: " . localtime() . " --- Tick count: " ,
Win32::GetTickCount(), "\n";
05. }
```

## Sleeping

Yes, it is true that Perl has a native `sleep()` function. However, that only lets a script fall asleep for blocks of 1-second intervals. This is much too slow when you are writing a time-sensitive service or daemon. This is where the `Win32::Sleep()` function comes in:

```
Win32::Sleep( $TimeInMilliseconds );
```

The only parameter it takes is the number of milliseconds to fall asleep.

When a Perl script calls `Win32::Sleep()`, the thread running the command will enter a wait state. This frees the CPU from having to process the Perl script until the thread wakes up. If your Perl script is constantly polling some service for changes to a variable or some hardware state, calling this function for a small duration of time can free up precious CPU time but still keep the script responsive.

## GetShortPathName

On Win32 platforms, paths can consist of long filenames. These names can consist of spaces and multiple "." characters. This is a drastic change from the DOS legacy of 8.3 filenames.

However, sometimes old DOS applications cannot handle such long filenames. Therefore, `Win32::GetShortPathName()` will convert any long pathname into a valid DOS-like 8.3 style of path.

```
Win32::GetShortPathName( $LongPath );
```

The only parameter is a long path string. The function will convert it to an appropriate short pathname that you can use with any DOS-like application that requires 8.3 file paths.

### Example 9.29 Examining various paths

```
01. my $UserProfilePath = $ENV{USERPROFILE};
02. my $ShortPath = Win32::GetShortPathName( $UserProfilePath );
03. my $LongPath = Win32::GetLongPathName( $ShortPath );
04. my( $Drive, $Path ) = Win32::GetFullPathName( $LongPath );
05. print "Original path: $UserProfilePath\n";
06. print "Short path: $ShortPath\n";
07. print "Long path: $LongPath\n";
08. print "\n";
09. print "Script name: $0\n";
10. my( $Path, $File ) = Win32::GetFullPathName( $0 );
11. print "Script path: ", Win32::GetFullPathName( $0 ), "\n";
12. print "Script directory: $Path\n";
13. print "Script file: $File\n";
```

## **GetLongPathName**

Just as `Win32::GetShortPathName()` returns the short version of a path, `Win32::GetLongPathName()` expands the short path to its long version.

```
Win32::GetLongPathName( $ShortPath );
```

The only parameter is a short path string. This can be obtained with a call to `Win32::GetShortPathName()`.

When this function is called, it compares the short pathname with the current directory tree. It is possible that the directory tree structure has changed since the last time the short path was obtained. In this case, it is possible that the long path might not be what was intended.

Refer to [Example 9.29](#) for an example of using this function.

## **GetFullPathName**

Unlike the other two `GetXXXXPathName()` functions, the `Win32::GetFullPathName()` function serves a different purpose. It is designed to compute the full path, not just convert between short and long pathnames.

```
Win32::GetFullPathName( $Path );
```

The only parameter is a path string. This can be a relative or absolute path.

This function will expand the name of a specified path into its full path. For example, if you pass in `".."`, the function will return the full path of the current directory.

If this function is called expecting an array result, two values are returned:

`$Result[0]` The path leading up to the specified file or directory

`$Result[1]` The file or directory name

This function does not examine the hard drive to determine its result. Therefore, you can specify paths that do not exist. The function will treat it as if it does exist and will calculate the appropriate results.

Refer to [Example 9.29](#) for an example of using this function.

## **Note**

*Don't bother trying to pass a string into `Win32::GetFullPathName()` that ends in a backslash. It will result in a Runtime Exception error. Therefore, a path like "c:\\" will cause your script to crash.*

## **CopyFile**

There used to be a time when the fastest way to copy a file was to open the file, set binary mode, read in data, write it, repeat until all data has been copied, and then close the file. It was pretty simple to do, but it ended up taking several lines of code. Win32 Perl has built in a file copy function that uses the Win32 API `CopyFile()` function.

```
Win32::CopyFile( $SourcePath, $DestPath, $fOverwrite );
```

The first parameter (`$SourcePath`) is a path to a valid existing file. This file will be copied. This can be either a relative or absolute path or UNC. This path cannot include any wildcards.

The second parameter (`$DestPath`) is a path to where the file will be copied. This can be either a relative or absolute path or UNC. This path cannot include any wildcards.

The third parameter (`$fOverwrite`) is a flag that indicates whether or not any existing file that `$DestPath` points to should be overwritten. If this value is `TRUE`, then any existing file will be overwritten. Otherwise, the copy process will fail if the destination file already exists.

If the file is successfully copied, the function returns `1`; otherwise, it returns `0`.

### **Example 9.30 Copying files**

```
01. my $SourcePath = Win32::GetFullPathName( $0 );
02. my $DestPath = $SourcePath . ".backup";
03. print "Copying $SourcePath\n";
04. print "to $DestPath...\n";
05. if( Win32::CopyFile( $SourcePath, $DestPath, 0 ) )
06. {
07.   print "Successfully copied!";
08. }
09. else
10. {
11.   print "Failed to copy. Error: $^E.\n";
12. }
```

## **Win32 Extension Functions**

Just like the built-in Win32 functions, there is another set of functions that exist in the Win32 namespace. However, this set is not built into Win32 Perl. These functions need to be loaded by explicitly loading the `Win32` extension.

To use these functions, a script must supply the following line:

```
use Win32;
```

This will load the extension and enable the script to call its functions.

### **Retrieving Microprocessor Information**

An undocumented function returns the type of microprocessor used by the computer. This function is `GetChipName()`:

```
Win32::GetChipName();
```

The return value is the type of the machine's microprocessor, such as `486` or `586`. (On a Pentium II NT box, this function returns `586`.)

This function should only be considered useful and accurate on a Windows 95 machine because it relies on data that Microsoft's Developer Network suggests should not be used on Windows NT machines (and it is provided with NT for backward compatibility with Windows 95). This does not reflect multiprocessor environments.

For more accurate processor information, use the `GetProcessorInfo()` function in the `Win32::AdminMisc` extension.

### ***Determining Platform Architecture***

The `GetArchName()` function returns a string that indicates the architecture of the machine running the Perl script. There was a time when this was useful because Windows NT could run on a DEC Alpha, MIPS, and PowerPC architecture in addition to standard Intel.

This function might become useful again now that Intel has released its 64-bit processor. Likewise, ports of Win32 Perl to Windows CE might find this function useful:

```
Win32::GetArchName();
```

The function returns an architecture string such as "x86" (referring to the Intel architecture). It is interesting to note that this function simply returns the same string as the environment variable `PROCESSOR_ARCHITECTURE`.

Therefore, a call to `Win32::GetArchName()` is functionally equivalent to `$ENV{PROCESSOR_ARCHITECTURE}`.

### ***Expanding Environment Strings***

The `%ENV` hash provides access to the system's environment variables. This is a useful hash, but it does not help too much when you need to expand a string with several environment variables. For example, a string such as

```
%ProgramFiles%\%USERDOMAIN%\%USERNAME%
```

has several environment strings that need to be expanded. It is possible to write Perl code that would parse each environment variable and resolve it using the `%ENV` hash, but it is far easier to call the `ExpandEnvironmentStrings()` function:

```
Win32::ExpandEnvironmentStrings( $String );
```

The only parameter is a text string. This may or may not have any environment strings that require expanding.

The function returns the expanded version of the specified string. If the string contains no environment strings, it is returned unmodified.

## ***Displaying Message Boxes***

A Perl script can easily display a simple GUI message box by calling the `Win32::MsgBox()` function:

```
Win32::MsgBox( $Message [, $Flags [, $Title ] ] );
```

The first parameter (`$Message`) is a message string that the message box will display.

The second and optional parameter (`$Flags`) indicates just how the message box is created. This value is the result of OR'ing a button flag (see [Table 9.17](#)), an icon flag (see [Table 9.18](#)), and a modality flag (see [Table 9.19](#)). None of these flags must be included; if any are left out, the default values will result. Examine each of the tables to determine which values are the defaults.

The third and optional parameter (`$Title`) specifies what the title of the message box will be.

The function will result in a graphical message box that the user must interact with. When the user has selected a button on the message box, the function returns.

The function will return a value indicating the user's interaction with the message box. The possible values are listed in [Table 9.20](#).

### ***Example 9.31 Displaying a message box***

```
01. use Win32;
02. my $Message = "Do you really want to know?";
03. my( $Path, $Script ) = Win32::GetFullPathName( $0 );
04. my $Result = Win32::MsgBox( $Message,
05.                           0x00000003 | 0x00000020 |
06.                           0x00001000,
07.                           $Script );
08. {
09.     print "Yes, you really want to know.\n";
10. }
11. elsif( 0x07 == $Result )
12. {
13.     print "No, you don't want to know.\n";
14. }
15. else
16. {
17.     print "Oh, just forget it.\n";
18. }
```

**Table 9.17. Message Box Button Flags**

Constant	Value	Description
MB_OK	0x00000000	The message box contains one push button: OK. This is the default.
MB_OKCANCEL	0x00000001	The message box contains two push buttons: OK and Cancel.
MB_ABORTRETRYIGNORE	0x00000002	The message box contains three push buttons: Abort, Retry, and Ignore.
MB_YESNOCANCEL	0x00000003	The message box contains three push buttons: Yes, No, and Cancel.
MB_YESNO	0x00000004	The message box contains two push buttons: Yes and No.
MB_RETRYCANCEL	0x00000005	The message box contains two push buttons: Retry and Cancel.
MB_CANCELTRYCONTINUE	0x00000006	Windows 2000 (and higher): The message box contains three push buttons: Cancel, Try Again, Continue. Use this message box type instead of MB_ABORTRETRYIGNORE.

**Table 9.18. Message Box Icon Flags**

Constant	Value	Description
MB_ICONSTOP	0x00000010	A stop-sign icon appears in the message box.
MB_ICONERROR		
MB_ICONHAND		
MB_ICONQUESTION	0x00000020	A question-mark icon appears in the message box.
MB_ICONEXCLAMATION	0x00000030	An exclamation-point icon appears in the message box.
MB_ICONWARNING		
MB_ICONINFORMATION	0x00000040	An icon consisting of a lowercase letter <i>i</i> in a circle appears in the message box.
MB_ICONASTERISK		

**Table 9.19. Message Box Modality Flags**

Constant	Value	Description
MB_APPLMODAL	0x00000000	The user must respond to the message box before continuing work. This is the default type unless specified otherwise.
MB_SYSTEMMODAL	0x00001000	The message box becomes a top-level window that cannot be covered by other windows.

**Table 9.20. Message Box Return Values**

Constant	Value	Description
IDOK	0x01	The OK button was selected.
IDCANCEL	0x02	The Cancel button was selected.
IDABORT	0x03	The Abort button was selected.
IDRETRY	0x04	The Retry button was selected.
IDIGNORE	0x05	The Ignore button was selected.
IDYES	0x06	The Yes button was selected.
IDNO	0x07	The No button was selected.
IDCLOSE	0x08	The Close button was selected.
IDHELP	0x09	The Help button was selected.
IDTRYAGAIN	0x0a	The Try Again button was selected.
IDCONTINUE	0x0b	The Continue button was selected.

### **Shutting Down a Machine**

If your user account has been granted the appropriate privileges, your script can force a Win32 machine to shut down using the `InitiateSystemShutdown()` function:

```
Win32::InitiateSystemShutdown( $Machine, $Message, $Count, $Force,  
$Reboot );
```

The parameters for the `InitiateSystemShutdown()` function are defined as follows:

- `$Machine` is the name of a machine that is to shut down. If this parameter is an empty string (" "), the local machine will be affected.
- `$Message` is a text string message that will be displayed on the screen of the computer specified in the first parameter. Typically, this is something informing the user that the machine is going to shut down and that all files should be saved.
- `$Count` is a numeric value that represents the amount of time (in seconds) before the shut down process begins—a type of countdown. When a shutdown is initiated, the message specified as the second parameter displays on the screen, and the number of seconds before shut down displays. This countdown will start at the time specified as the third parameter. An administrator will usually specify some amount of time, such as five minutes, giving a user enough time to properly finish saving files or finish and send email messages.
- `$Force` is a flag indicating whether the machine should force a shutdown. If a system shutdown is forced, applications will not be given a chance to save open files—they are just shut down. This forceful shutdown will occur if the fourth parameter is a nonzero value. Otherwise, each application will ask the user to save open files during the shutdown process.
- `$Reboot` is a flag that indicates whether to reboot the machine. If the value is nonzero, the machine will shut down followed by a reboot.

The `InitiateSystemShutdown()` function returns `TRUE` (1) if the specified computer will initiate the shutdown process; otherwise, the computer will not shut down, and the function returns `FALSE` (0).

For the `Win32::InitiateSystemShutdown()` function to be successful, the user who calls the function must have the privilege `SE_SHUTDOWN_NAME` ("force a shutdown from a remote system") enabled on the computer to be shut down.

If the `$Count` value (the third parameter) is zero (0), no message will be displayed on the specified computer, and the shutdown will begin immediately—without any chance to abort the shutdown.

### **Aborting System Shutdown**

At any time during the countdown process (before shutdown actually begins), the shutdown can be aborted using the `Win32::AbortShutdown()` function:

```
Win32::AbortSystemShutdown( $Machine );
```

The only parameter (`$Machine`) specifies which computer is to abort a shutdown. If the parameter is an empty string (" "), the local machine is specified.

This function will return `TRUE` (1) if the function is successful. The function is considered successful if it submits the request to abort a shutdown and the specified machine accepts the request. Even if the specified machine is not shutting down but the abort request was successfully sent, the function is considered successful. Otherwise, the function fails and returns `FALSE` (0).

Just as with `Win32::InitiateSystemShutdown()`, the `SE_SHUTDOWN_NAME` privilege ("force a shutdown from a remote system") must be enabled on the specified machine for the user who calls the `Win32::AbortShutdown()` function.

If a user initiates a system shutdown by means of pressing `Ctrl+Alt+Del` and specifying "shutdown," the operating system will initiate the shutdown with a countdown of zero; therefore, this type of shutdown cannot be aborted using this function.

### **Example 9.32 Shutting down a system**

```
01. use Win32;
02. my $Machine = shift @ARGV || Win32::NodeName();
03. my $Count = 50;
04. my $DelayTime = 5000;
05. my $Message = "The system is going to shut down in $Count
seconds";
06. my $fForce = 0;
07. my $fReboot = 1;
08. print "Initiating shutdown on $Machine...";
09. if( Win32::InitiateSystemShutdown( $Machine, $Message,
10.                               $Count, $fForce,
$fReboot ) )
11. {
12.     print "successful.\nPausing for $DelayTime milliseconds.\n";
13.     Win32::Sleep( $DelayTime );
```

```

14.     print "Aborting shutdown... ";
15.     if( Win32::AbortSystemShutdown( $Machine ) )
16.     {
17.         print "successfully aborted.\n";
18.     }
19. else
20. {
21.     print "Oops! Failed to abort the shutdown.\n";
22.     print Win32::FormatMessage( Win32::GetLastError() ), "\n";
23. }
24. }
25. else
26. {
27.     print "Failed to initiate a shutdown.\n";
28.     print Win32::FormatMessage( Win32::GetLastError() ), "\n";
29. }

```

### ***Exiting Windows Instead of System Shutdown***

There is an alternative to the `Win32::InitiateSystemShutdown()` function that provides for more options, although it only works on the local machine. This is the `Win32::AdminMisc::ExitWindows()` function:

```
Win32::AdminMisc::ExitWindows( $ExitType );
```

The only parameter (`$ExitType`) is a flag that specifies exactly how Windows will exit. This can be any value from [Table 9.21](#). This value can also be logically OR'ed with `EWX_FORCE`, which will cause applications to quit without saving any open files. This is a harsh way of shutting down, but it could be necessary if an unattended shutdown or logoff is required.

***Table 9.21. Various Exit Types for the `Win32::AdminMisc::ExitWindows()` Function***

<b>Exit Type</b>	<b>Description</b>
<code>EWX_LOGOFF</code>	Log the user off. Applications will be informed, so the user might be prompted to save files.
<code>EWX_POWEROFF</code>	Force the system to shut down and power off. The system must support poweroff. On Windows NT, the user account calling the function must have the <code>SE_SHUTDOWN_NAME</code> privilege.
<code>EWX_REBOOT</code>	Shut down the system and reboot the computer. On Windows NT, the user account calling the function must have the <code>SE_SHUTDOWN_NAME</code> privilege.
<code>EWX_SHUTDOWN</code>	Shut down the system but don't reboot. On Windows NT, the user account calling the function must have the <code>SE_SHUTDOWN_NAME</code> privilege.

If the `Win32::AdminMisc::ExitWindows()` function is successful, it returns a `true` value and begins its exiting process; otherwise, it fails and returns `false`.

## **Resolving User SIDs**

At first glance, one would think that the following two functions belong in [Chapter 3](#) where user accounts are discussed. These functions are the `Win32::LookupAccountName()` and `Win32::LookupAccountSID()` functions. [Chapter 3](#), however, refers to managing groups and accounts, whereas these functions do not manage anything. As a matter of fact, I find it difficult to determine exactly where they belong because they are more related to topics beyond the scope of this book (and are probably more suited to a discussion about Windows NT security). These functions either look up a security identifier (SID) for an account or look up an account for a given SID. You will find it useful to understand exactly what a SID and an account are.

An account is just a group of information that relates to a given user, such as username, password, privileges, and so forth. Accounts, however, are not limited only to users. Every machine registered as a member of a domain has an account. So does every local and global user group. Additionally, every primary domain controller (PDC) of a trusted domain has an account.

A SID is a binary data structure that NT's local security authority (or LSA, the heart of Windows NT security) maintains for each account. When an administrator sets permissions on a file or directory, he selects a user or group and grants it a particular type of permission. Every account has only one SID, and every SID is mapped to only one account.

For all practical purposes, a SID will not be of any use to a Perl script for several reasons. The most glaring reason is that you can't do anything with a SID. Because a SID is an internal Win32 data structure, it does not do much good to expose it to Perl. To quote from Ralph Davis's *Win32 Network Programming* (Addison Wesley Developers Press), "[SIDs are] intended to be an opaque data type, which applications are not supposed to access directly." Because SIDs are mapped to accounts, it is much easier to specify a user's account or group name instead of managing a large binary data structure.

Now that SIDs have been briefly explained, it is time to examine the `LookupAccountName()` function:

```
Win32::LookupAccountName( $Machine, $Account, $Domain, $Sid,
$Type );
```

The parameters for the `LookupAccountName()` function are defined as follows:

- `$Machine` is the name of the computer that is to perform the user account lookup. If this parameter is an empty string (""), it assumes to use the local machine. This value can a machine name like `SERVER_A`, or it can be a proper computer name such as `\SERVER_A`.
- `$Account` is the name of the account to be looked up. This can be a username, a group name, a trusted domain name, or a computer name (the computer and domain names must end with a dollar sign).
- `$Domain` is an output parameter, so it must be a scalar variable and will be set by the function; therefore, it can be any value when calling the function (as long as it is a scalar variable and not a constant). If the function is successful, this scalar is set with the name of the domain to which the account belongs.
- `$Sid` is an output parameter, so it must be a scalar variable. This variable will be set to the value of the account's SID. Note that this SID is not a text string. It is a binary SID in absolute format. See [Chapter 11](#), "Security," for more information on binary SIDs.

- `$Type` is an output parameter, so it must be a scalar variable. This variable will be set to a value that represents the type of SID that this account represents. The value that this variable is set to can be any one of the values listed in [Table 9.22](#).

**Table 9.22. Different SID and Account Types**

Value Account	Type
0x01	User account. The account (and SID) represents a typical user account.
0x02	Group account. The account (and SID) represents a global group.
0x03	Domain account. The account (and SID) represents a domain account. This is also known as a domain trust account that is used by one domain to log on to another domain. Examples are <code>ACCOUNTING</code> , <code>REDMOND</code> , and <code>MY_DOMAIN</code> .
0x04	An alias account. This is just the formal Win32 description for a local group account. Examples are <code>ADMINISTRATORS</code> , <code>USERS</code> , and <code>GUESTS</code> .
0x05	Well-known name account. These are those accounts that you always see but can never modify such as <code>SYSTEM</code> , <code>EVERYONE</code> , <code>CREATOR OWNER</code> .
0x06	Deleted account. This represents an account that has been deleted.
0x07	Invalid account. The account has become corrupt or somehow invalid.
0x08	Unknown. The system is unable to determine any information regarding the SID or account.

If the `LookupAccountName()` function is successful, it will map an account name from the specified machine to its domain name, SID, and type and will return a `true` value. Otherwise, it returns `false`.

After you have a SID, you can figure out which account it represents by using the `LookupAccountSID()` function:

```
Win32::LookupAccountSID( $Machine, $Sid, $Account, $Domain,
$Type );
```

The parameters for the `LookupAccountSID()` function are defined as follows:

- `$Machine` is the name of the computer that is to perform the user account lookup. If this parameter is an empty string (""), the `LookupAccountSID()` function assumes that the local machine is performing the lookup. This value can be a machine name like `SERVER_A`, or it can be a proper computer name like `\SERVER_A`.
- `$Sid` is an account's SID. This is typically obtained using the `Win32::LookupAccountName()` function. Note that this SID is not a text string. It is a binary SID in absolute format. See [Chapter 11](#) for more information on binary SIDs.
- `$Account` is an output parameter, so it must be a scalar variable. This is the name of the account that the `$Sid` represents.

- `$Domain` is an output parameter, so it must be scalar and will be set by the function; therefore, it can be any value when calling the function (as long as it is a scalar variable and not a constant). If the function is successful, this scalar is set with the name of the domain in which the account was found.
- `$Type` is an output parameter, so it must be a scalar variable. This variable will be set to a value that represents the type of account that the SID represents. The value that this variable is set to can be any one of the values previously listed in [Table 9.22](#).

If the `LookupAccountSID()` function is successful, it maps a SID from the specified machine to its account name, domain name, and type and returns a `true` value. Otherwise, it returns `false`.

It is important to note that for both the `LookupAccountName()` and `LookupAccountSID()` functions, the `$sid` is a binary data structure that by itself is of no use. This structure is usually something like 400 bytes long and does not provide any information that is easily discernable. Only those who have a specific need for this data will find it useful. [Example 9.33](#) demonstrates a very simplistic use of `LookupAccountName()` and `LookupAccountSID()`.

### ***Example 9.33 Using Win32::LookupAccountName() and Win32::LookupAccountSID()***

```

01. use Win32;
02. my @Sids;
03. my @AccountTypes = (
04.   'an invalid',
05.   'a user',
06.   'a global group',
07.   'a domain',
08.   'a local group',
09.   'a well-known',
10.   'a deleted',
11.   'an invalid',
12.   'an unknown'
13. );
14. my @Accounts = (
15.   'system', 'creator owner', 'guest',
16.   'administrator', Win32::LoginName()
17. );
18. my $iCount = 0;
19. print "\nLookup Accounts by their account names:\n";
20. foreach my $Account ( sort( @Accounts ) )
21. {
22.   my( $Domain, $Sid, $SidType );
23.   if( Win32::LookupAccountName( '', $Account, $Domain, $Sid,
$SidType ) )
24.   {
25.     print ++$iCount, " ) $Account is $AccountTypes[$SidType]
account..\n";
26.     push( @Sids, $Sid );
27.   }
28.   else
29.   {
30.     print "Could not look up the \u$Account account. Error:
$^E\n";

```

```

31.    }
32. }
33. print "\nLookup Accounts by their SIDs:\n";
34. $iCount = 0;
35. foreach my $Sid ( @Sids )
36. {
37.   my( $Account, $Domain, $SidType );
38.   if( Win32::LookupAccountSID( '', $Sid, $Account, $Domain,
$SidType ) )
39.   {
40.     print ++$iCount, " ) \u$Account is $AccountTypes[$SidType]
account..\n";
41.   }
42. }

```

## Summary

This chapter covered some of the more obscure extensions that have functions of significance. These functions are not well understood. For programmers who need to use them, however, they are indispensable.

The `Win32::Console` extension provides a script with the capability to manage console applications. This is quite handy when an administrator wants an application that requires user input but does not need the overhead of a GUI interface. Additionally, it is very handy for a service or CGI script to kick open a console and output information to it.

The `Win32::Sound` extension provides a coder with rudimentary sound capabilities. Most of the scripts I write to act as services use this extension to play back `.WAV` files, which tell me what event is occurring. A most valuable tool!

Considering that most of the Win32 Perl users are not familiar with creating their own proprietary extensions, the `Win32::API` extension is a god-send. Providing the capability to access most of the Win32 API functions, a script can access those functions that a public extension has yet to address.

Finally, the miscellaneous functions found in the `Win32` extension are often overlooked but are a treasure chest of capability.

Perl's wide range of `Win32` extensions makes it a remarkable scripting tool. Even the vast number of extensions cannot cover all programmers' needs, however. This is where Perl's capability to extend itself comes in. [Chapter 10](#), "Writing Your Own Extension," discusses how you can create your own extension to solve problems that no other extension solves.

# Chapter 10. Writing Your Own Extension

Everything has limits, and Perl is no exception. Although it is true that Perl has a very rich function set, there are things it just cannot do.

If you need to do some task very quickly, for example, Perl might not be able to run fast enough for your needs. Or you might not be able to control your machine in a particular way with Perl. This is why extensions are written.

Because so many extensions have been written, most people never need to write their own extension; instead, you can use one that someone else made. Usually people upload their extensions to the Comprehensive Perl Archive Network (CPAN), where you can get them whenever you need them (<http://www.cpan.org/>).

Sometimes, however, even the extraordinary CPAN does not have what you need. This is when you need to consider making your own extension. It might sound quite daunting, but it really isn't.

This chapter describes the basics behind the inner workings of Perl and how to apply that knowledge to create your very own Perl extension. For those who do not intend to write extensions, this chapter is useful in understanding how existing extensions work. Many bugs have been caught not by C++ programmers but by Perl hacks who were just looking over an extension's source code.

This chapter requires at least a little knowledge of C or C++; otherwise, you might be scratching your head in confusion. Likewise, this chapter is only a brief introduction into creating extensions. It is meant for coders who need to extend Perl quickly and is by no means the definitive guide to making extensions. For that I refer you to either the Perl man pages (that come with ActivePerl) or some other sources such as the excellent work *Advanced Perl Programming* by Sriram Srinivasan (O'Reilly & Associates).

## What Is an Extension?

Perl has the capability to load external libraries of functions used to augment Perl's built-in functions. Different operating systems use different methods, but the Win32 platform uses dynamic link libraries (DLLs) or, as the ActiveState version of Win32 Perl prefers, Perl link libraries (PLls).

These DLLs do not differ from the other DLLs you have on your system, such as `KERNEL32.DLL`, `MSVCRT20.DLL`, or `VBRUN300.DLL`. They are basically a collection of functions compiled and stored in one file. Really, this is the same thing as a Perl module because it too is a collection of functions all stored in one file. The difference between a Perl extension and a Perl module is that the module contains functions written in Perl and the extension contains functions written in another language, typically C or C++. Additionally, extensions rely on modules to load the extension into Perl.

Because an extension is written in C (from this point on, I will assume you are using C/C++ to write the extension), it is capable of anything that C is capable of doing—and that is an awful lot!

Think about all the available programs written in C, such as administration utilities, word processors, and games. The capabilities of these C-based applications are quite extensive; your extension can be also. An extension can make calls into the Win32 API or load up other libraries. An example of this could be that an extension could load a faxing library and generate a fax that could be sent out over a fax/modem. Likewise, you could write an extension that makes a connection to an Internet FTP site that then downloads a file. Although you can do the same thing using Perl functions, you might prefer an extension because of the speed increase that an extension gives and/or source code protection issues. Everything covered in this book is either an extension or related to a Win32 Perl extension.

### Note

*API* is an acronym for application programming interface. APIs are everywhere and are necessary for programmers. An API is the set of rules that a programmer must use to interact with some software. If you want to have Windows open a file, for example, you would use the Win32 API's `CreateFile()` function (if you open a file using C's `fopen()`, it will, in turn, call `CreateFile()`). To successfully use `CreateFile()`, you must pass in certain values in a particular order. Therefore, you must know the syntax and rules for calling the function. Likewise, to understand the return value (which usually determines whether the function was successful), you need a list of return codes. This information is part of the Win32 API. If it were not for APIs, nobody would know how to tell Windows to open a file. It might seem obvious that APIs need to be publicly available, but there was a time when some companies, such as IBM, would not publish their APIs. This way, they could charge a fee to those who wanted to use them. Even today, some APIs are not easily obtained, such as Windows NT's LSA API (the Local Security Authority API, which is the set of functions that manipulates very low-level security).

## What Is a DLL?

In the Win32 world, a DLL is just a collection of functions. Such a collection of functions is not necessarily simple, however. A DLL can be as simple as a single function, or it can be as complex as being a self-contained program. All Win32 Perl extensions are DLLs, so it is important to understand how they work. They are discussed in detail later in this chapter in the section "[The DllMain\(\) Function](#)."

## How to Write a Perl Extension

Before understanding how to create an extension, you need to familiarize yourself with a couple of rather important topics, the first of which is how Perl handles variables.

### The Guts of Perl Variables

A Perl scalar variable is really a memory structure known as an SV (scalar variable). When your script creates a new scalar variable, a memory structure is allocated, and the value is assigned to it. If you ever access the value in a different context (such as accessing an integer as a string), the SV will convert itself to the context value. If you create a variable, assigning it the value `10`, for example, SV is created containing the integer value `10`. If you then try to print out the value, the SV will have to convert the integer `10` into the text string "10". At that point, the SV will store the text string; therefore, it will then have two values: `10` and "10". If you later try to add .5 to the variable, the SV will have to convert the value of `10` to a floating point of `10.0`. This new value will also be stored so that three values are associated with the value: an integer, a string, and a floating-point number.

It is important to note that when a string representation of a value is stored in an SV structure, it is stored not as a character string—as a C programmer might expect—but as a byte array. Both the number of bytes (or characters—the length of the string) as well as the data are stored in the structure. This is important because a Perl string (unlike a C char string) can have NULL characters (characters with a value of `0`) within them. You can use a scalar variable to represent a text string (like "Hello world!") or binary data (like a GIF file or any other binary data). In C and C++, a character string is NUL-terminated, meaning that the end of the string is represented by a `0` value. A Perl string does not work this way. It works in a fashion more like Pascal, where there is a string of bytes and a numeric value that tells how many bytes long the string is. Taking this into

consideration, it might be inaccurate to refer to Perl strings. They are more like byte arrays, a collection of bytes that can be of any value.

Because a Perl string, also known as a PV (pointer variable), can be binary data, it is possible to place a C structure into a string. This is essentially what the Perl `pack()` function does. It is rather easy for an extension to pack a C structure (or multiple structures) into a PV and return it back to Perl where it can be `unpacked()`'ed. [Example 10.24](#) illustrates this later in this chapter.

Therefore, every time you create a Perl scalar variable, you are really allocating an SV memory structure.

## Creating SVs

It is rather easy to create an SV. All you need to do is figure out which kind of SV to create. When you first create the SV, it must know which type of value it is being assigned. [Table 10.1](#) lists the different SV value types.

**Table 10.1. Different SV Value Types**

Value Type	Description
PV	A string such as "Hello World". PV stands for pointer value. This can also be used to represent binary byte arrays and C structures.
IV	An integer value. This is a number large enough to hold a pointer. Typically, this means that it is 32 bits long; depending on your machine, however, it could be 32 or 64 bits long.
NV	This is equivalent to a double value.
RV	A reference to another SV.

An SV is created by using the `newSVxx()` macros defined in the Perl header files (discussed later in this chapter in the section "[Beginning Your Extension](#)"). The `newSVxx()` macros include the following:

```
01.SV* newSViv( IV lValue )
02.SV* newSVnv( double dValue )
03.SV* newSVsv( SV* pSV )
04.SV* newSVpv( const char* szString, int iLength )
```

The result of each of these macros is a new SV pointer; otherwise, if the macro failed for some reason, the return value is `NULL`.

These macros are defined as follows:

- **`newSViv()`**. Creates an integer-based SV with the value passed into the macro. Notice that the IV data type is 32 bits (or 64 bits on a 64-bit machine) in length, longer than an int.
- **`newSVnv()`**. Accepts a double value.

- `newSVsv()`. Accepts a pointer to another SV. This will really just make a clone or copy of the existing SV passed into it.
- `newSvpv()`. Requires a little bit of explaining. The first parameter passed into it (`const char* szString`) is a pointer to an array of bytes. Notice that I specify an array of bytes, not characters. This is because this array is not necessarily a character string; it could be a data buffer. Perl does not expect that this string will be null terminated, as a proper C string is.

The second parameter to `newSvpv()` (`int iLength`) is the length of the array to which the first parameter points. This length should not include any terminating null. It is common to use the result of C's `strlen()` function as the value for the second parameter because it returns only the number of characters in the string and does not include the terminating null character. If this parameter is `0` or the constant `na`, the macro will calculate the length of the string using `strlen()`.

In [Example 10.1](#), several SVs are created.

### ***Example 10.1 Creating several different SVs***

```
01. #include <perl.h>
02.
03. long      lValue = 3287663;
04. double    dValue = 3.141592;
05. char      *pszValue = "This is a test string.";
06. char      *pBuffer = {1, 0, 53, 255, 0, 32};
07. int       iLength = 6;
08. SV        *pSV[5];
09.
10. pSV[0] = newSViv( lValue );
11. pSV[1] = newSVnv( dValue );
12. pSV[2] = newSvpv( pszValue, strlen(pszValue) );
13. pSV[3] = newSvpv( pBuffer, iLength );
14. pSV[4] = newSVsv( (SV*) pSV[1] );
```

Notice that the last line makes a copy of the second SV that was created.

### ***Determining the SV Type***

If you are unsure which type an SV is, you can query it to discover the kind of data it holds. The `SVTYPE()` macro located in `lib\CORE\sv.h` does this:

```
enum SVTYPE( SV* pSV )
```

The parameter passed into the macro (`pSV`) is a pointer to an SV.

The `SVTYPE()` macro will return an enumerated value that represents a type of value. The enumerated types are defined in the header files discussed later in the section "[Beginning Your Extension](#)." This macro is the equivalent of using Perl's `ref()` function. The return value is one of the types listed in [Table 10.2](#).

**Table 10.2. Values Returned From the `SvTYPE()` Macro**

Value	Description
<code>SVt_IV</code>	Integer.
<code>SVT_NV</code>	Double.
<code>SVt_PV</code>	String.
<code>SVt_PAV</code>	Array.
<code>SVt_PVHV</code>	Hash.
<code>SVt_PVCV</code>	A code segment that represents a Perl subroutine. This is the equivalent of the value of <code>\&amp;MySubRoutine()</code> . Use of this type in an extension is beyond the scope of this book.
<code>SVt_PVGV</code>	A glob. This type represents a namespace. This is equivalent to Perl's <code>*variable</code> name feature. Use of this type in an extension is beyond the scope of this book.
<code>SVt_RV</code>	A reference to another type.
<code>SVt_NULL</code>	An undefined value. This is similar to the result of <code>undef</code> .
<code>SVt_PVMG</code>	A magical or blessed SV. This is a special SV that has "magical" powers. It acts either as a Perl object or as a "smart" variable such as <code>\$!</code> and <code>\$^E</code> , which returns either an error number or an error text string, depending on the context used. Use of this type in an extension is beyond the scope of this book.

When you stumble on an SV and don't know what type of data it represents, you can use the `SvTYPE()` macro to determine how to handle it, as in [Example 10.2](#). Note that lines 23 through 26 are using `printf()` to print the string data obtained in line 22. This works fine if the SV contains only text data, but because an SV can contain binary data, including `NUL` characters, `printf()` might not be the best way to print its contents.

### **Example 10.2 Dumping the contents of an SV**

```

01. #include <sv.h>
02.
03. static PerlInterpreter *my_perl;
04.
05. void DumpSV( SV* pSV )
06. {
07.     switch( SvTYPE( pSV ) )
08.     {
09.         case SVt_IV:
10.             printf( "The SV is an integer with a value of %ld.\n",
11.                     (long) SvIV( pSV ) );
12.             break;
13.
14.         case SVt_NV:
15.             printf( "The SV is an floating point with a value of
%lf.\n",
16.                     (double) SvNV( pSV ) );

```

```

17.     break;
18.
19.     case SVt_PV:
20.     {
21.         unsigned int uiLength = 0;
22.         char *pszString = SvPV( pSV, uiLength );
23.         printf( "The SV is an string with a length of %d and "
24.                 "a value of '%s'.\n",
25.                 uiLength,
26.                 pszString );
27.     }
28.     break;
29.
30.     default:
31.         printf( "Unable to determine the SV's type.\n" );
32.     }
33.     return;
34. }

```

### **Verifying an SV Type**

When you have an SV, you can guarantee that the SV is of a particular type by using the `SvxOK()` macros located in `<lib\CORE\sv.h>`. Some of them are as follows:

```

int SvIOK( SV* pSV )
int SvNOK( SV* pSV )
int SvPOK( SV* pSV )
int SvROK( SV* pSV )
int SvOK( SV* pSV )
int SvTRUE( SV* pSV )

```

The only parameter that all these macros accepts is a pointer to an SV structure.

Each macro will return a `1` if the specified SV contains a value of the specified type and a `0` if it does not. [Table 10.3](#) explains which macro checks for what type of value.

**Table 10.3. The `SvxOK()` Macros and What They Check**

Macro	Functionality
<code>SvIOK()</code>	Checks for an integer value.
<code>SvNOK()</code>	Checks for a double value.
<code>SvPOK()</code>	Checks for a string value.
<code>SvROK()</code>	Checks for a reference value. References are discussed in the " <a href="#">References</a> " section later in this chapter.
<code>SvOK()</code>	Checks that any value is present. If no value is present (as in <code>undef</code> ), <code>FALSE</code> is returned.

**Table 10.3. The SvxOK() Macros and What They Check**

Macro	Functionality
SvTRUE()	Checks that the value is <code>true</code> —that it does not equate to <code>0</code> or <code>undef</code> .

### **Extracting Values from SVs**

After you have an SV, you can discover the value it holds by using the `SvxV()` macros:

```
IV SvIV( SV* pSV )
double SvNV( SV* pSV )
char* SvPV( SV* pSV, unsigned int uiLength )
SV* SvRV( SV* pSV )
```

All these macros take only one parameter, which is a pointer to an SV. The exception to this is the `SvPV()` macro, which also requires that a pointer to an integer be passed in as a second parameter. If `SvPV()` is successful, it will return a char pointer to the `Svs` character string. Additionally, the integer whose address is passed in as the second parameter is set to the length of the string.

If you use one of these macros on an SV of a different type, the SV will convert its value to the correct type. If an SV contains an integer but you use `SvPV()` on it, for example, the SV will create an internal character string version of the integer value and will return a pointer to that string.

If one of these macros cannot convert the SV's data type to the specified type, the macro returns `0`. If an SV contains the string `Hello world!`, for example, the `SvIV()` macro will return `0` because there is no appropriate way to convert the text string into an integer. If the string were `321`, it could be converted into the integer value `321`, so the `SvIV()` macro would return the value `321`.

The `SvIV()` macro will return the integer value from the specified SV.

The `SvNV()` macro will return the double value from the SV.

`SvPV()` will return a pointer to the character string held in the SV. The integer that the second parameter points to will be set to the length of the string.

The `SvRV()` macro will return a pointer to an SV to which the reference passed in points. It is important that any pointer passed into this macro actually be a reference to an SV and not an SV pointer. Use the `SVTYPE()` or `SvROK()` macros to ensure that you truly have a reference. References are explained later in this chapter in the section "References."

### **Setting SV Values**

When you need to set the value of an SV, you can use the `sv_setxv()` macros:

```
sv_setiv( SV* pSV, int iValue)
sv_setnv( SV* pSV, double dValue)
sv_setpv( SV* pSV, const char* pszString )
```

```
sv_setpvn SV* pSV, const char* pszString, int iLength )
sv_setsv( SV* pDestinationSV, SV* pSourceSV )
```

All these macros take a first parameter, which is a pointer to the SV whose value will be set. The macros are defined as follows:

- **sv\_setiv()**. Sets the specified SV with the specified integer value.
- **sv\_setnv()**. Sets the specified SV with the specified double value.
- **sv\_setpv()**. Copies the string pointed to by the second parameter into the SV. The string must be a NULL-terminated string.
- **sv\_setpvn()**. The same as **sv\_setpv()** except that it accepts a third parameter, which specifies how long the string is (in bytes). This enables you to specify a string of data that has embedded **NULL** characters.
- **sv\_setsv()**. Copies the source SV passed in as the second parameter to the destination SV passed in as the first parameter. The resulting destination SV (the first parameter) will be an identical copy of the source SV (the second parameter).

## Note

*Both the **sv\_setpv()** and **sv\_setpvn()** macros make copies of the strings passed into them. The memory whose pointer is passed into the macros can later be freed without any impact to the SV.*

The sample code in [Example 10.3](#) shows how to set SV values. Notice that the SV is created as an integer-based IV (line 6). Line 7 then changes the value of the SV. Finally, line 8 sets yet another value, but this time it will also translate the SV from an integer-based SV to a string-based SV. The point here is that an SV can change from type to type on the fly.

### **Example 10.3 Setting the value of an SV**

```
01. #include <embed.h>
02. #include <proto.h>
03.
04. static PerlInterpreter *my_perl;
05.
06. SV* pSV = newSViv( 4567 );
07.     sv_setiv( pSV, 1234567 );
08.     sv_setpv( pSV, "Hello Joe!" );
```

## **Concatenating Strings**

In addition to setting string values on SVs with **sv\_setpv()** and **sv\_setpvn()**, you can concatenate byte arrays (or strings) using the **sv\_catpvx()** macros:

```
void sv_catpv( SV* pSV, const char* pszString )
void sv_catpvn( SV* pSV, const char* pData, int iLength )
```

Both macros take an SV pointer as their first parameter and take a pointer to a string for the second parameter. The string in `sv_catpv()` must be NULL-terminated. [Example 10.4](#) describes how to concatenate two strings.

`sv_catpvn()` takes a third parameter, which is the length of the string. Its data (second parameter) can be any byte array. (It does not have to be a NULL-terminated string.)

[Example 10.4](#) illustrates the use of `sv_catpvn()` to concatenate byte arrays.

#### **Example 10.4 Concatenating strings**

```
01. #include <embed.h>
02. #include <proto.h>
03.
04. static PerlInterpreter *my_perl;
05.
06. char pszString1 = "This is a ";
07. char pszString2 = "concatenation test!";
08. unsigned int uiLength;
09. SV* pSV = newSVpvn( pszString1, strlen(pszString1) );
10. sv_catpvn( pSV, pszString2 );
11. printf( "%s", SvPV( pSV, uiLength ) );
```

The XS function `ExtensionGetFileSizes()`, defined later in [Example 10.27](#), searches for all files that match the specified criteria. For each file found, a memory structure is filled out that holds the file's name and size. Line 155 in [Example 10.27](#) concatenates this structure into a PV using `sv_catpvn()`. A Perl script that calls the `Win32::Test::GetFileSizes()` function will need to use `unpack()` to access the structures. [Example 10.5](#) shows how to unpack the data structures returned by a call to the sample `GetFileSizes()` function.

#### **Example 10.5 Unpacking concatenated data structures from the extension in Example 10.4**

```
01. use Win32::Test;
02. $Count = Win32::Test::GetFileSizes( "c:\\temp\\*.*", $Data );
03. %List = ( unpack( "A256L" x $Count, $Data ) );
```

#### **Reference Counts**

When an SV is created, it contains a reference counter that is set to 1. This counter indicates how many other structures, such as references (RVs are covered in the next section, "[References](#)"), point to this memory structure. If you create a reference that points to an SV, for example, the SV's reference counter increases by one. If you later attempt to delete the SV, the reference count decreases by one. Only when the reference count is 0 is it actually purged from memory.

This prevents some very bad situations, such as an RV that points to an SV that has been deleted. Any attempt to dereference the RV under this circumstance could be quite catastrophic because the reference is pointing to memory that no longer is a valid SV; it could be pointing to anything! Bad; very, very bad.

A reference count can be manually increased or decreased, but you are strongly discouraged from doing so unless you know what you are doing. If you need to force an SV to deallocate itself from memory, you could decrease the reference count until it is 0 (at which time Perl will automatically deallocate the memory structure). Likewise, if you want to guarantee that an SV is not allocated, you could manually increase the reference count. The `svREFCNT_xxx()` macros located in `lib\CORE\sv.h` manage reference counts:

```
svREFCNT_inc( SV* pSV )
svREFCNT_dec( SV* pSV )
```

The macros will either increase or decrease the reference count of the SV passed into it. If the reference count is decreased to 0, the SV is purged from memory.

## References

A special type of SV is known as a reference. References are the same as the Perl reference `$Ref` in [Example 10.6](#).

### **Example 10.6 Using a reference**

```
01. my $Dozen = 12;
02. my $Ref = \$Dozen;
03. print "There are $$Ref in a dozen.";
```

Running this code will print out the string `There are 12 in a dozen`. When you create a Perl reference, a new SV is created and marked as a reference. This SV will point to another SV structure.

When the variable `$Dozen` is assigned its value, an SV structure is allocated in memory. Then, when the script creates the variable `$Ref`, Perl allocates yet another SV structure. Because this second SV is going to be assigned the reference to the `$Dozen` variable (that is, what `$Dozen` represents), Perl will set a flag in `$Ref`'s SV structure that indicates that it is a reference to another SV. Then Perl will set a pointer in `$Ref`'s SV structure to point to `$Dozen`'s SV.

To write the equivalent of [Example 10.6](#) in C, you would make an int and assign it the value 12. Then you make an integer pointer and assign it the address of your first integer. You then print out your string as in [Example 10.7](#).

### **Example 10.7 The C equivalent of Example 10.6**

```
01. int iDozen=12;
02. int *piRef;
03.
04. piRef = &iDozen;
05. printf( "There are %d in a dozen.", *piRef );
```

This might sound rather vague and complicated, but it isn't. Consider that for every scalar variable you make in Perl, you are creating an SV somewhere in memory. If you create a reference variable, you are still creating an SV, but no scalar value will be stored in it. Instead, a pointer to another SV

will be stored. This allows Perl to dereference the SV and find the original SV structure that has the value printed in [Example 10.6](#).

To create a reference in an extension, you can use the `newRV()` macros:

```
SV* newRV( SV* )
SV* newRV_inc( SV* )
SV* newRV_noinc( SV* )
```

All these methods return a pointer to an SV structure. This SV structure will, in turn, point to another SV. Normally, when you create a reference, the SV you are referring to will increase its reference counter by one. All these macros except `newRV_noinc()` will do this. Actually, `newRV()` and `newRV_inc()` perform the same function; there is no difference between the two macros. Generally speaking, `newRV_noinc()` is not used except for some specific reason to not increase the reference count. Most all references will increase the reference count.

All the `newRV()` macros accept an SV pointer, which is what the new reference will point to, as illustrated in [Example 10.8](#). This example is not as clear as if we were doing this using straight C (rather than using Perl SVs) as in [Example 10.7](#). This is because, to illustrate how RVs are used, we need to create an RV and then immediately dereference it. This defeats creating an RV altogether, but it does illustrate how one is created and dereferenced. Line 9 creates the reference to `pSV`. Line 12 then dereferences `pRefSV` (using the `sv()` macro, which is discussed later). Note that after line 12, `pDerefedSV` is equal to `pSV`. Finally, line 15 extracts the integer value from the SV and prints it out in line 16.

### ***Example 10.8 Creating a reference to an SV***

```
01. #include <proto.h>
02.
03. static PerlInterpreter *my_perl;
04.
05. // First create an SV to hold the integer 12...
06. SV* pSV = newSViv( 12 );
07.
08. // Now create a reference SV (an RV) which points to pSV...
09. SV* pRefSV = newRV( pSV );
10.
11. // Now we must dereference the reference to get the original
SV...
12. SV* pDerefedSV = SV( pRefSV );
13.
14. // Last we get the integer value from the dereferenced SV...
15. int iDozen = SvIV( pDerefedSV );
16. printf( "There are %d in a dozen.", iDozen );
```

### **Warning**

*The lifetime of an SV is defined by its reference count. As long as the count is greater than 0, the SV is alive. Because of this, it is really important that SVs not be created on the stack—unless your*

*code is aware that stack-based variables are only available in the scope of a given code block. Consider this code:*

```
SV *MyBadFunction( )
{
    SV svBad;
    SV *pReference = newRV( &svBad );
    return( pReference );
}
```

*The problem here is that the SV `svBad` is an automatic variable (created on the stack). The function returns a reference to `svBad`; when the function returns, however, `svBad` is automatically destroyed. The returned reference does not know that `svBad` has been destroyed, however. Any attempt to dereference the returned SV pointer will most likely cause a fault.*

## **The SV Lifespan**

When the reference count of an SV is decremented to `0`, the SV is automatically destroyed and purged from memory. This defines the lifespan of an SV. If you create an SV, it remains in memory until you either manually remove it or decrease its reference count to `0` (in which case Perl will remove it). This can be a problem if you need to create an SV that you return to Perl from some function. Suppose, for example, you write a function that adds two numbers together and creates an SV to hold the result. If you want to pass this SV back to Perl so that your script can use it, you might run into a slight problem. What if no variable catches your return result?

In this example, assume that your Perl script calls your function like this:

```
$Result = AddNumbers( 1, 2 );
```

The `AddNumbers()` function will add `1` and `2` together and will then create an SV that holds the result. The SV is then passed back to Perl so that `$Result` will use it. When the SV is created, the reference count is `1`. When the SV is assigned to `$Result`, the reference count will increase to `2`. Later, if `$Result` is destroyed, the reference count is reduced back to `1`, but because it is not `0`, it is not purged from memory. This means you might have SVs hanging around with reference counts of `1` that are just taking up memory. This is not a good thing.

What you can do is tag the SV as mortal. When an SV is mortal, Perl will keep track of it, and when it leaves the Perl script's current scope, its reference count will automatically be decreased. If you tag the SV created in the `AddNumbers()` function as mortal, its reference count will decrease by one after it leaves the `AddNumbers()` function (the scope in which it was created). If some variable, such as `$Result`, links to it, the reference count is increased to `1` again, and it is not destroyed. When `$Result` is destroyed, however, the reference count goes to `0`, and hence the SV is also destroyed.

Mortality just guarantees that an SV that has been created will be destroyed if it is no longer used.

To indicate mortality, you can use one of the mortal macros located in `lib\CORE\proto.h`:

```
SV* sv_newmortal()
SV* sv_2mortal( SV* pSV )
SV* sv_mortalcopy( SV* pSV )
```

These macros are defined as follows:

- **sv\_newmortal()**. Creates a new SV tagged as mortal. You can use this SV as you would any other, but at the end of the current scope, it will be destroyed.
- **sv\_2mortal()**. Marks an existing SV passed into the macro as mortal.
- **sv\_mortalcopy()**. Creates a copy of the SV passed in and marks that new SV as mortal.

All the mortal macros return a pointer to a new SV that has been marked as mortal. The exception to this is **sv\_2mortal()**, which does not create a new SV but just marks the SV passed in as mortal and returns a pointer to it.

If an SV has been marked as mortal, you can manually increase its reference count by one to essentially remove its mortality. This means that when the SV leaves the current Perl scope, its reference count is decremented (but from 2 to 1). Because the count does not reach 0, it is not purged from memory.

## Perl and Arrays

When you deal with an array, you are playing with a special type of SV known as an AV (array variable—not to be confused with an access violation error). Simply stated, an AV is a structure that contains a list of pointers to other SV-like structures. These other structures can be other SV-related structures such as scalars (SV), arrays (AV), hashes (HV), and references (RV).

### ***Creating AVs***

To create a new array, you use the **newAV()** macro:

```
AV* newAV();
```

A pointer to the AV structure is returned.

An alternative to **newAV()** is the **av\_make()** macro. This will create an array and populate it, assuming you already have an array of SV pointers:

```
AV* av_make( long lNumber, SV** pSVList )
```

The first parameter (**int iNumber**) is the number of elements that will be placed into the array.

The second parameter (**SV\*\* pSVList**) is a pointer to an array of SV pointers.

This will return a pointer to the newly created AV, as depicted in [Example 10.9](#).

### ***Example 10.9 Creating AVs***

```
01. #include < proto.h>
```

```

02.
03. static PerlInterpreter *my_perl;
04.
05. AV* pAV1 = newAV();
06. AV* pAV2 = NULL;
07. SV* ppSV[ 10 ];
08. int iTemp;
09. for( iTemp = 0; iTemp < 10; iTemp++ )
10. {
11. ppSV[ iTemp ] = newSViv( iTemp );
12. }
13. pAV2 = av_make( 10, ppSV );
14. printf( "There are %ld elements in pAV2.\n", av_len( pAV2 ) );

```

After you have an AV, you can query it to find out how many elements it contains using the `av_len()` macro:

```
long av_len( AV* pAV )
```

The `av_len()` macro takes a pointer to an AV and returns the number of elements that the AV contains.

## **Array Management**

When you have an AV, you can add elements to and remove elements from it by using the `av` series of macros. The first of these macros is `av_clear()`, which will clear out the contents of the array:

```
void av_clear( AV* pAV )
```

The only parameter is a pointer to the AV. When this macro is used, the reference count for every element in the array will be decremented by one. After this decrement, each element is replaced with the `undef` value. The overall effect is that the array is cleared—that is, no more values are left in the array. The array, however, remains the same size.

There is no return value.

Similar to `av_clear()` is the `av_undef()` macro, which not only will decrement each element's reference count, it will also decrement the array's reference count.

```
void av_undef( AV* pAV )
```

This macro does not return any value. As with any SV, if an element's reference count is decremented to `0`, it is deallocated from memory automatically. Similarly, if the array's reference count is decremented to `0`, it is freed up from memory.

If you just want to remove all entries from the array, use `av_clear()`; if you need to destroy an array, however, call `av_undef()`.

You can put elements into an array as well as remove them from the array. This is where the `av_push()` and `av_pop()` macros come in:

```
01. void av_push( AV* pAV, SV* pSV )
02. SV* av_pop( AV* pAV )
```

The first parameter for both of these macros (`AV*pAV`) is a pointer to the AV.

The `av_push()` accepts one additional parameter (`SV* pSV`), which is a pointer to an SV that is to be pushed onto the end of the array. After this has been done, the number of elements in the array increases by one, and the SV pointer passed in becomes the last element in the array. This macro does not return any value.

The `av_pop()` macro will return a pointer to the SV that has been popped off of the array. The SV pointer returned points to the SV that was the last element of the array. After the macro is called, the second to last element becomes the last element, and the number of array elements decreases by one.

The reference count of the SV removed from the array by the `av_pop()` macro is not altered by the pop action. This is important because if your code calls `av_pop()` but ignores its return value, the popped SV will not be destroyed (because its reference count is not altered). Let's say your code pops off 100 stack elements but ignores the return value. Unless somehow the SVs popped off of the array are destroyed, they will be sitting around taking up memory. [Example 10.10](#) shows two functions that pop all elements off of an array. The first one, `RemoveElementsBad()`, pops off all elements from the array but ignores them. `RemoveElementsGood()` marks each SV popped off the array as mortal. This guarantees that the SV will be destroyed by Perl.

### ***Example 10.10 Popping elements from an array***

```
01. #include <proto.h>
02. static PerlInterpreter *my_perl;
03.
04. void RemoveElementsBad( AV* pAV )
05. {
06.     int iCount = av_len( pAV );
07.     while( iCount-- )
08.     {
09.         av_pop( pAV );
10.     }
12. }
13.
14. void RemoveElementsGood( AV* pAV )
15. {
16.     int iCount = sv_len( pAV );
17.     while( iCount-- )
18.     {
19.         sv_2mortal( av_pop( pAV ) );
20.     }
21. }
```

Two additional macros perform similar functions to `av_push()` and `av_pop()` ; these are called `av_unshift()` and `av_shift()`:

```
void av_unshift( AV* pAV, long lNumber )
SV* av_shift( AV* pAV )
```

Both macros accept a first parameter (`pAV`), which is a pointer to the AV to be processed.

The `av_unshift()` accepts a second parameter (`lNumber`), which is the number of elements to put in front of the array, starting at element `0`. Each element added to the array contains the `undef` value. This macro is handy if you want to grow the array size without having to add SVs.

The `av_shift()` is just like `av_pop()` except that the first element is removed from the array. The return value is a pointer to the element that has been removed from the array. The array is decremented by one. Just like `av_pop()`, the removed SV's reference count is not altered.

Only two more macros allow for array management. These are the `av_fetch()` and `av_store()` macros:

```
SV** av_store( AV* pAV, long lIndex, SV* pSV )
SV** av_fetch( AV* pAV, long lIndex, long lValue )
```

Both macros accept three parameters. The first (`AV* pAV`) is a pointer to an AV that will be processed.

The second parameter (`long lIndex`) is the index number of the element to be either stored or retrieved.

For the `av_store()` macro, the third parameter (`SV* pSV`) is a pointer to an SV to be stored into the element specified in parameter two.

The `av_fetch()` macro accepts a third value; if it is a nonzero value, the specified array element will be replaced with `undef`. This is equivalent to calling `av_fetch()` followed by `av_store()`, in which you are storing an empty SV in the same array element. Whereas this is not a common practice, it can be useful if you are just removing an element from an array.

Both `av_store()` and `av_fetch()` return a pointer to the SV that is the element of the array specified by the second parameter. [Example 10.11](#) shows how these macros can be used.

### ***Example 10.11 Manipulating array elements***

```
01. #include <perl.h>
02. #include <proto.h>
03.
04. void DumpArray( AV *pAV );
05.
06. static PerlInterpreter *my_perl;
07.
08. SV* pSV;
09. SV* pSV2;
```

```

10. AV* pAV = newAV( );
11. int iTemp;
12.
13. // Add ten elements to the array
14. for( iTemp = 0; iTemp < 10; iTemp++ )
15. {
16.     pSV = newSViv( iTemp );
17.     av_push( pAV, pSV );
18. }
19.
20. // pAV now contains: [(end) 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
21. (front)]
22. printf("The array contains:\n");
23. DumpArray( pAV );
24.
25. // Remove the first array element (9)
26. pSV = av_pop( pAV );
27.
28. // pSV now == 9
29. // pAV now contains: [8, 7, 6, 5, 4, 3, 2, 1, 0]
30.
31. // Take the last element and move it to the first element
32. pSV2 = av_shift( pAV );
33.
34. // pAV now contains: [8, 7, 6, 5, 4, 3, 2, 1]
35. av_push( pAV, pSV2 );
36.
37. // pAV now contains: [0, 8, 7, 6, 5, 4, 3, 2, 1]
38. // Add an empty element to the end of the array
39. av_unshift( pAV, 1 );
40.
41. // pAV now contains: [0, 8, 7, 6, 5, 4, 3, 2, 1, undef]
42. // Store the previously removed element at the end
43. iTemp = (int) av_len( pAV );
44.
45. // Decrease the index by one since the array is zero based
46. // (the first element starts at position 0)
47. printf( "The array is now:\n" );
48. DumpArray( pAV );
49.
50. void main(int argc, char **argv, char **env)
51. {
52.     my_perl = perl_alloc();
53.     perl_construct(my_perl);
54.     perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
55.     perl_run(my_perl);
56.     perl_destruct(my_perl);
57.     perl_free(my_perl);
58. }
59.
60. void DumpArray(AV *pAV)

```

```

61. {
62.     long lCount = av_len( pAV );
63.     long lTemp;
64.     unsigned int uiLength;
65.     for( lTemp = 0; lTemp < lCount; lTemp++ )
66.     {
67.         SV **pSVTemp = av_fetch( pAV, lCount, 0 );
68.         printf( "\tElement %d: '%s'\n", lTemp, SvPV(*pSVTemp,
uiLength) );
69.     }
70. }
```

## Hash Variables

Perl provides the capability to create arrays with associated keys—that is, each element in the array has a name associated with it. This way, it is not necessary to keep track of which element number contains what data inside an array. Instead, you always refer to the array's element by name, not by number. This is called an *associated array*, commonly known as a *hash*.

### ***Creating Hashes***

A hash is created by using the `newHV()` macro defined in `lib\CORE\proto.h`:

```
HV* newHV();
```

The macro will create a new hash and return a pointer to it. Unlike arrays, this is the only macro that can create a hash.

### ***Storing Hash Values***

After you have a hash, you can store values into it using the `hv_store()` macro defined in `lib\CORE\proto.h`:

```
SV** hv_store( HV* pHV, const char* pszKey, unsigned long
ulKeyLength, SV*
pValue, unsigned long ulHash )
```

The first parameter (`HV* pHV`) is a pointer to the hash structure into which you are storing the value.

The second parameter (`const char* pszKey`) is a character string (or a byte array) that represents the key with which the value will be associated.

The third parameter (`unsigned long ulKeyLength`) is the length of the string passed in as the second parameter. You *must* pass in a value here because the macro will not figure it out for you.

The fourth parameter (`SV* pValue`) is a pointer to an SV that you are storing.

The fifth, and last, parameter (`unsigned long ulHash`) is the hash to use. If you pass in a `0`, the macro will create a hash for you. You should pass in `0` unless you really know what you are doing.

If the `hv_store()` macro is successful, it returns a pointer to the SV. A pointer to the SV is stored into the hash. This is important to understand because if the SV is destroyed, the hash entry points to an invalid SV.

## **Retrieving Hash Values**

To retrieve a value from a hash, you use the `hv_fetch()` macro:

```
SV** hv_fetch( HV* pHV, const char* pszKey, unsigned long
ulKeyLength, long lVal
)
```

The parameters for the `hv_fetch()` macro are defined as follows:

- `HV* pHV` is a pointer to the HV.
- `const char* pszKey` is a pointer to a character string that is the key.
- `unsigned long ulKeyLength` is the length of the key string passed in as the second parameter. You need to specify this because the macro will not figure it out for you.
- `long lVal` is used internally, and you should always specify a `0` for this value.

If the `hv_fetch()` macro is successful, it returns a pointer to a pointer to an SV. If the macro fails, it returns `NULL`.

Because the return value is a double pointer, you must make sure you dereference the pointer before using it, as in [Example 10.12](#).

### **Example 10.12 Fetching a key's value from a hash**

```
01. static PerlInterpreter *my_perl;
02. //...assuming that pHV is an existing hash...
03. SV** ppSV;
04. char* pszKey = "name";
05. ppSV = hv_fetch( pHV, pszKey, strlen( pszKey ), 0 );
06. if( NULL != ppSV )
07. {
08.     unsigned int uiLength;
09.     SV* pSV = *ppSV;
10.     char* pszValue = SvPV( pSV, uiLength );
11.     printf( "The value of the %s key is: %s.\n", pszKey,
pszValue );
12. }
```

## **Verifying Key Existence in a Hash**

To check whether a key exists in a hash, you can use the `hv_exists()` macro:

```
bool hv_exists( HV* pHV, const char* pszKey, unsigned long ulKeyLength )
```

The parameters for the `hv_exists()` macro are defined as follows:

- `HV* pHV` is a pointer to the hash.
- `const char* pszKey` is a pointer to a string that is the key name.
- `unsigned long ulKeyLength` is the length of the key specified in the second parameter.

If the key exists, the `hv_exists()` macro will return `TRUE`; otherwise, it returns `FALSE`.

### ***Deleting Keys from a Hash***

To delete a key from a hash, you use the `hv_delete()` macro:

```
SV *hv_delete( HV* pHV, const char* pszKey, unsigned long ulKeyLength, long lFlag )
```

The parameters for the `hv_delete()` macro are defined as follows:

- `HV* pHV` is a pointer to the hash.
- `const char* pszKey` is a pointer to a string that is the key name.
- `unsigned long ulKeyLength` is the length of the key specified in the second parameter.
- `long lFlag` is a flag. If this flag is set to `G_DISCARD`, the key will be deleted and nothing is returned. Otherwise, the key is deleted, but a mortal copy of the SV that the key pointed to is created and returned.

### ***Removing Data from a Hash***

Two other macros exist to remove data from a hash, the `hv_clear()` and `hv_undef()` macros:

```
void hv_clear( HV* pHV )
void hv_undef( HV* pHV )
```

Both of these macros require that a pointer to a hash be passed in.

The `hv_clear()` will clear out all the entries in the hash, but the hash will remain. The reference count for each SV that the hash's elements point to is decremented. Then each element in the hash is removed. This does not explicitly destroy the SV in each element. (If their count becomes zero, however, Perl destroys them automatically.) Just as with `av_clear()`, the elements are removed, but the hash itself remains (its reference count is not touched). After the `hv_clear()` macro is called, the hash is empty.

The `hv_undef()` will, like `hv_clear()`, clear out all the entries, but it will also destroy the hash. This causes all elements' reference counts to be decremented. Note that this will not explicitly destroy any of the SVs. (If their reference counts become zero, however, Perl then destroys them.)

Unlike `hv_clear()`, `hv_undef()` decrements the hash's reference count. If the reference count becomes zero, the hash is destroyed.

## ***Walking Through a Hash***

Walking through a hash is not as simple as one would like, but it really is not too difficult either. Basically, if you need to enumerate each key in a hash, you must follow these steps:

1. Prepare the hash for a walkthrough using the `hv_iterinit()` macro.
2. Locate the first element by calling the `hv_iternext()` macro. The first time this macro is called, it will point to the first element in the hash.
3. Retrieve the current element's key and value with the `hv_iterkey()` and `hv_iterval()` macros, respectively.
4. Repeat steps 2 and 3 until either all elements have been exhausted or the desired element has been found.

An alternative to steps 2 and 3 is to call the `hv_iternextsv()` macro.

To start a hash walkthrough, you need to first prepare the hash to be iterated with the `hv_iterinit()` macro:

```
long hv_iterinit( HV* pHV )
```

The `hv_iterinit()` macro takes in one parameter, which is a pointer to a hash. The macro will prepare the HV to be iterated. It also returns the number of keys in the hash.

After the hash has been prepared for iteration, you need to use the `hv_iternext()` macro:

```
HE* hv_iternext( HV* pHV )
```

The only parameter is a pointer to the hash you are iterating.

The `hv_iternext()` macro will return a pointer to a data structure known as an HE (a hash element). This is a pointer to a link in a linked list that the HV uses to store its keys and values. This pointer is used in other macros.

Now that you have iterated and have an `HE` pointer, you can use it to retrieve both the key and the value of the hash's element using the `hv_iterkey()` and `hv_iterval()` macros:

```
char* hv_iterkey( HE* pHE, long* plKeyLength )
SV* hv_iterval( HV* pHV, HE* pHE )
```

The `hv_iterkey()` macro accepts the `HE` pointer returned from the `hv_iternext()` macro as a first parameter.

The second parameter of `hv_iterkey()` (`plKeyLength`) is a pointer to a long. The length of the key name will be set into this long. The macro returns a pointer to a character string, which is the name of the key.

The `hv_iterval()` macro accepts a pointer to the hash as the first parameter and the HE pointer returned by the `hv_itternext()` macro as the second parameter.

The `hv_iterval()` macro will return a pointer to the SV that holds the value associated with the particular element in the hash.

Now that you have retrieved both the key and the value of the iterated hash element, you can call `hv_itternext()` again to move on to the next element in the hash.

Just for the sake of convenience, you can use the `hv_itternextsv()` macro. This macro iterates the next element in the hash and retrieves the key and value at the same time:

```
SV* hv_itternextsv( HV* pHV, char** ppszKey, long* plKeyLength );
```

The parameters for the `hv_itternextsv()` macro are defined as follows:

- `HV* pHV` is a pointer to the hash you are iterating.
- `char** ppszKey` is a pointer to a pointer to the element's key name.
- `long* plKeyLength` is a pointer to a long that will be stored with the length of the key name.

The `hv_itternextsv()` macro will advance to the next element in the hash and will then return a pointer to the key's value's SV.

## Warning

*When walking through a hash, it is safe to delete the current element using `hv_delete()`, but it is not safe to store anything in the hash (using `hv_store()`). The reason is that when you store elements in a hash, the hash might need to reorganize its internal hash. This would break the capability to continue walking through the hash.*

*If `hv_store()` was called, your code would have to call `hv_itterinit()` and start the walkthrough all over.*

Example 10.13 illustrates all the iteration macros. The example creates two functions—`DumpHash()` and `DumpHash2()`. The former uses the `hv_itternext()` technique, and the latter uses `hv_itternextsv()`. Notice that in both functions, `pSVValue` points to the element's value SV. In lines 19 and 35, the `SvPV()` macro is used to convert the SV to a character string. This makes it easier to print out because we then do not have to be concerned with whether the value is an IV, NV, or PV.

### **Example 10.13 Iterating a hash**

```
01. #include <extern.h>
02. #include <perl.h>
03. #include <proto.h>
04.
05. static PerlInterpreter *my_perl;
```

```

06.
07. void DumpHash( HV *pHV )
08. {
09.     HE *pHE = NULL;
10.    long lElements = hv_iterinit( pHV );
11.    printf( "Dumping %d elements:\n", lElements );
12.    printf( "\tKey\tValue\n\t--\t----\n" );
13.    unsigned int uiLength = 0;
14.    while( NULL != ( pHE = hv_iternext( pHV ) ) )
15.    {
16.        long lKeyLength;
17.        char *pszKey = hv_iterkey( pHE, &lKeyLength );
18.        SV *pSVValue = hv_iterval( pHV, pHE );
19.        printf( "\t'%s'\t'%s'\n", pszKey, SvPV( pSVValue,
uiLength ) );
20.    }
21. }
22.
23. void DumpHash2( HV *pHV )
24. {
25.    long lKeyLength;
26.    char *pszKey = NULL;
27.    SV *pSVValue = NULL;
28.    unsigned int uiLength = 0;
29.    long lElements = hv_iterinit( pHV );
30.    printf( "Dumping %d elements:\n", lElements );
31.    printf( "\tKey\tValue\n\t--\t----\n" );
32.
33.    while( NULL != ( pSVValue = hv_iternextsv( pHV, &pszKey,
&lKeyLength ) ) )
34.    {
35.        printf( "\t'%s'\t'%s'\n", pszKey, SvPV( pSVValue,
uiLength ) );
36.    }
37. }
38.
39. void main(int argc, char **argv, char **env)
40. {
41.     my_perl = perl_alloc();
42.     perl_construct(my_perl);
43.     perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
44.     perl_run(my_perl);
45.     perl_destruct(my_perl);
46.     perl_free(my_perl);
47. }
```

## Beginning Your Extension

The overview of an extension is quite simple. It consists of the following elements:

- Any required headers including the necessary Perl headers

- Any required non-Perl functions
- Any Perl functions that require the use of the `XS()` macro
- The Perl bootstrap function
- Optionally, the Windows `DllMain()` function

The very first thing you should do in your extension source file is include the headers. Some Perl headers are required and need to be included. In [Example 10.14](#), line 14 declares that lines 14 to 22 are to be compiled as regular C and not as C++ (by using `extern "C"`). Otherwise, you will see some pretty nasty function name mangling, which can be the cause of many frustrated debugging sessions. This is only necessary if you are using C++ with either the ActiveState headers or the headers for Perl 5.005 (and later) and the `PERL_OBJECT` macro is defined.

With ActiveState's Perl version 5.003 and Perl 5.005 (and higher), you can make use of the `CPerl` class. This requires C++ to compile, and it provides many nifty capabilities. Basically, this class abstracts the Perl interpreter into an object. All the Perl functions are mapped to methods of the object. Starting with version 5.005, the `CPerl` class has been renamed to `CPerlObj` (but they are effectively the same).

## Tip

*If you are compiling the extension using the ActiveState headers, you need to define the following three macros:*

```
#define MSWIN32
#define EMBED
#define PERL_OBJECT
```

*Without these macros defined (they can be empty but need to be defined), the build process will fail. In my extensions, I test for the definition of `PERL_OBJECT`. If it is not defined, I assume that I am using the core distribution's headers; otherwise, I assume that I am using ActiveState's.*

*I usually define these macros in the MSVC project settings dialog box. This way, they are not hard coded in my files anywhere, and I can create different configurations with or without these macros, enabling me to create configurations that compile using the core distribution's headers or the ActiveState headers.*

*Starting with Perl 5.6 (a.k.a. version 5.006), many changes have been made. The version of Perl that ActiveState distributes defines the following macros instead of the ones previously specified:*

```
#define EMBED
#define MSWIN32
#define HAVE_DES_FCRYPT
#define MULTIPLICITY
#define PERL_IMPLICIT_CONTEXT
#define PERL_IMPLICIT_SYS
#define PERL_IMPLICIT_CONTEXT
#define PERL_NO_GET_CONTEXT
#define PERL_POLLUTE
#define USEITHREADS
```

Notice that the `PERL_OBJECT` macro is not defined in this group. This is because the default Win32 build of Perl (called ActivePerl) does not define the macro when it is compiled by the folks at ActiveState.

To use the `CPerl/CPerlObj` class, a macro called `PERL_OBJECT` must be defined. The macro can be empty, but it needs to be defined. This can be defined in the build settings of a Microsoft Visual C++ project, or it can be hard wired into the source files.

It is very important to understand that an extension compiled with `PERL_OBJECT` will not work with Perl that was not compiled with the macro defined and vice versa.

Notice that, in [Example 10.14](#), there are some checks for Borland and Microsoft compilers. This has been borrowed from Gurusamy's code; he was the first to point this out to me, and I use it religiously now.

### **Example 10.14 Including the required Perl headers**

```
01. //The following code should be in the beginning of every
extension you write
02. #ifdef __BORLANDC__
03.   // wchar_t is in tchar.h but unavailable unless UNICODE is
defined
04.   typedef wchar_t wctype_t;
05. #endif
06.
07. #include <windows.h>
08. // Gurusamy's right, Borland compilers need this here!
09. #include <stdio.h>
10. // Gurusamy's right, MS compilers need this here!
11. #include <math.h>
12.
13. #if defined(__cplusplus) && !defined(PERL_OBJECT)
14. extern "C" {
15. #endif
16.
17. #include "EXTERN.h"
18. #include "perl.h"
19. #include "Xsub.h"
20.
21. #if defined(__cplusplus) && !defined(PERL_OBJECT)
22. }
23. #endif
```

### **Warning**

Take note that Microsoft introduced an incompatibility problem with its MSVC Service Pack 4 (SP4). Due to some alignment issues Perl extensions compiled using SP4 are not binary compatible with Perl that was compiled with pre-SP4 and vice versa. ActiveState intends to use SP3 for

*compiling Perl v5.6 until the next version of Perl is released (which will most likely break binary compatibility anyway).*

*So if you compile an extension using MSVC and it always causes a runtime error when loading, check that you do not have SP4 installed.*

## Writing Extension Functions

When you start writing the C/C++ functions that you want to be able to access from within Perl, you need to use the `XS()` macro:

```
XS( FunctionName )
```

The `XS()` macro declares the function as a Perl function. The parameter passed in will be the function name that will be mapped in the bootstrap function (see the section "[The Bootstrap Function](#)" later in this chapter).

Really, this is quite easy to do. Whenever you are creating a function that will be called directly from a Perl script, you just declare the function using the `XS()` macro. This will guarantee that the function can communicate with Perl.

After you have declared your function, you need to set up the Perl variables that are needed by many of Perl's macros. This is accomplished using the `dXSARGS` macro:

```
dXSARGS
```

That is it, nothing more. This will set up your Perl stacks and create the `sp` (stack pointer) variable in addition to a series of other things. Basically, the `dXSARGS` macro is what declares everything your function needs to get values passed into the function from Perl and to set variables to be passed back to Perl. `dXSARGS` also initializes some variables that are used by Perl's macros and can be used by your function.

## The Stack

Perl makes use of several stacks, but one stack requires a bit of explanation. When a Perl script calls an extension function, any values passed into the function are placed on a stack. The extension's function is given access to this stack, which it can use for both input and output between the extension's function and Perl. This stack is just an array of SV pointers. Each pointer represents a passed-in parameter. This stack is the equivalent to a Perl subroutine's `@_` array.

When a function calls the `dXSARGS` macro, the function is given access to the stack, and a variable called `items` is created. The `items` variable contains a value indicating how many parameters were passed into the function (just like `argc`).

## Retrieving Values Passed into the Function

To retrieve the passed-in parameters from the stack, you can use the `ST()` macro:

```
SV* ST( int iElement )
```

The `ST()` macro accepts an integer that identifies which element of the stack you are requesting. The return value is a pointer to an SV.

Make sure that you do not specify an element that is greater than or equal to `items` because `items` is the total number of elements on the stack and the stack is a zero-based array. This means that if `items` is 1 (indicating only one value on the stack), the one parameter is in element 0 of the stack. This would be accessed with `ST(0)`. Because there is no second value, a call to `ST(1)` would be a bad idea.

After an SV pointer has been retrieved from the stack, its value can be retrieved using one of the value extraction macros described in the previous section "Extracting Values from SVs."

## Returning Values

After the function is ready to end and return to Perl, it can place return values onto the stack so that Perl can retrieve them. The same stack that stored the input parameters will be used for the return values.

When assigning values back to the stack, it is best to tag them as mortal. This way, if the value is ignored, on the function's return to Perl, the SV will die and be purged from memory (for an explanation of this, refer to the earlier section "The SV Lifespan").

To assign return values back to the stack, you use the same `ST()` macro that you used to extract the values from the stack. It takes the following format:

```
ST( iStackElement ) = pSV;
```

`iStackElement` is the stack element number, and `pSV` is a pointer to an SV. This will just assign the specified SV pointer to the specified stack element.

Refer to [Example 10.15](#) for an illustration of using the `ST()` macro.

After you have prepared the stack with your return values, you need to tell Perl how many values you have stored on the stack using the `XSRETURN()` macro:

```
void XSRETURN( long lValue )
```

The parameter passed into the macro indicates how many values are stored on the stack. If the value 4 is passed into this macro, Perl will pop off the first four elements from the stack when the function returns. The calling script can retrieve these values:

```
($Value1, $Value2, $Value3, $Value4) = Win32::Test::MyFunction();
```

If you have only one or no values to return, you might as well use the `XSRETURN_xxx()` macros:

```

XSRETURN_NO
XSRETURN_YES
XSRETURN_UNDEF
XSRETURN_EMPTY
XSRETURN_IV( long lValue )
XSRETURN_NV( double dValue )
XSRETURN_PV( char* pszValue )
XSRETURN_PVN( char* pszValue, int iLength )

```

These macros set the stack with the specified value and report the correct `XSRETURN()` value. Your code does not need to call either `ST()` or `XSRETURN()` if you use one of these macros; however, it works if you have only one or no values to return.

The `XSRETURN_xxx()` macros will return one value back to Perl. The value returned depends on the macro used. [Table 10.4](#) lists the returned values.

Note that the `XSRETURN_PV()` macro accepts only a character string and does not accept a length. This means the macro requires that the string passed in be a NULL-terminated string.

**Table 10.4. Return Value Types**

Return Value	Description
<code>XSRETURN_NO</code>	Returns a <code>FALSE</code> value (0).
<code>XSRETURN_YES</code>	Returns a <code>TRUE</code> value (nonzero).
<code>XSRETURN_UNDEF</code>	Returns <code>undef</code> .
<code>XSRETURN_EMPTY</code>	Returns nothing. This is the same as <code>XSRETURN(0)</code> .
<code>XSRETURN_IV()</code>	Returns the long value that is passed in.
<code>XSRETURN_NV()</code>	Returns the double value that is passed in.
<code>XSRETURN_PV()</code>	Returns the string that is passed in. This string must be a NULL-terminated string.

The `XSRETURN_xxx()` macros work by creating a mortal SV and putting it as the first element of the stack. It then uses the `XSRETURN(1)` macro, indicating that Perl can expect one value on the stack.

[Example 10.15](#) sets two return values on the stack and returns, informing Perl to expect two return values. Notice that line 5 creates a new string-based SV, and line 6 first marks the SV as mortal and then stores it in the stack (as the first element—element 0). Line 7 creates a new SV, which is a floating-point number, marks it as mortal, and then stores it as the second element in the stack.

#### **Example 10.15 Returning two values from a function**

```

01. XS( MyFunction )
02. {
03.     dXSARGS;
04.     char *pszTest = "My test string!";

```

```

05.     SV* pSV = newSVpv( pszTest, strlen(pszTest) );
06.     ST( 0 ) = sv_2mortal( pSV );
07.     ST( 1 ) = sv_2mortal( newSVnv( (double) 6.02 ) );
08.     XSRETURN( 2 );
09. }
```

Both of the returned SVs were marked as mortal so that they would die after the function returns back to Perl. If the calling Perl scripts assign these return values to variables upon return of the function, their reference counts are increased from 0 to 1. This will prevent them from being automatically destroyed by Perl. If the script discards these values (does not assign the return values to any variable), however, they will indeed be destroyed.

[Example 10.16](#) demonstrates using the `ST()` macro to obtain values passed into the function from the calling Perl script. Notice that line 4 checks the `items` variable (which was initialized with the `dXSARGS` macro) to see whether two values were indeed passed in. Line 9 adds the integer values of both passed-in SVs, and line 10 uses the `XSRETURN_IV()` macro to tell Perl that it is returning one long value. If the two parameters are not passed into the function, line 14 tells Perl to return `undef`.

### ***Example 10.16 Returning one value from a function***

```

01. XS( MyFunction2 )
02. {
03.     dXSARGS;
04.     if( 2 == items )
05.     {
06.         long lSum;
07.         SV* pSV1 = ST( 0 );
08.         SV* pSV2 = ST( 1 );
09.         lSum = SvIV( pSV1 ) + SvIV( pSV2 );
10.         XSRETURN_IV( lSum );
11.     }
12.     else
13.     {
14.         XSRETURN_UNDEF;
15.     }
16. }
```

The Perl script that calls this can expect two return values:

```
($Text, $AvagadrosNum) = Win32::Test::MyFunction();
```

## **Modifying Input Parameters**

You might notice that some extensions change input variables. You might see a Perl script call some function such as this, for example:

```

01. if( Win32::Test::MyFunction( $Value1, $Value2, $Sum ) )
02. {
03.     print "The sum of $Value1 and $Value2 is $Sum.\n";
```

```
04. }
```

When the function is called, it might add the first two values passed in and store the sum in the third parameter. This is actually a very easy thing to do.

In your extension's function, all you need to do is retrieve the SV for the particular called parameter. In the preceding example, you want to set the value of the third parameter so that you need to get the SV for the third parameters using the `ST(3)` macro.

When the function is entered, the stack has three SV pointers on it. You want to modify the value of the third one. Therefore, you grab the SV pointer to it with this:

```
SV *pSV = ST( 3 )
```

This gives you an SV pointer to the Perl script's `$Sum` variable. When you set the value of this SV pointer, you are directly setting the value of `$Sum`. However, when you later use

```
ST( 3 ) = pSV;
```

you are just replacing the contents of the third element of the stack.

Simply stated, there is a difference between replacing the value of a stack element and replacing the value of an SV that happens to be pointed to by a stack element. [Example 10.17](#) illustrates this.

### ***Example 10.17 Changing the value of an input parameter***

```
01. XS( MyFunction )
02. {
03.     dXSARGS;
04.     long lValue1, lValue2, lSum;
05.
06.     if( items != 3 )
07.     {
08.         XSRETURN_NO;
09.     }
10.     lValue1 = SvIV( ST( 0 ) );
11.     lValue2 = SvIV( ST( 1 ) );
12.     lSum = lValue1 + lValue2;
13.
14.     // Get the third parameter's SV
15.     SV* pSV = ST( 3 );
16.     // Now pSV points to the SV that was passed in as the
17.     // 3rd parameter.
18.
19.     // Set the new value of the SV we got. This will literally
change
20.     // the contents of the Perl script's $Sum variable.
21.     sv_setiv( pSV, (long) lSum );
```

```

23. // Return a TRUE (non zero) value to indicate that the
24. // function was successful.
25. XSRETURN_YES;
26. }

```

## Calling Other Functions

Calling other functions from within a function that Perl calls directly might need some explaining. Normally, you would just call another function, such as a Win32 API call or a library function of some sort. If your function will make use of Perl macros, however, you need to treat it like an extension function.

Suppose your extension's function will call another function that processes some variable. If it needs to access any Perl macros, it must use the `dXSARGS` macro (just like your extension function). This will set up the Perl stack and the `items` variable as well. Now, that was not so difficult.

The difficulty with this is that the way you prototype your function depends on which version of Win32 Perl you are using. If you are building this extension for the core distribution (v5.005), you have no more work to do. If you are using the ActiveState version, however, you have to consider the `CPerl` object.

You see, when the ActiveState version of Perl (v5.003 and v5.006) calls into an extension's function, one of the parameters that is passed in is a pointer to the C++ Perl object. This object is an instance of the `CPerl` class (in Perl 5.005, it has been renamed to `cPerlObj`), of which all macros will make use. Generally, this is no problem and is invisible. However, when you call into another function, you need to pass in a pointer to that `CPerl` object; otherwise, the macros used in that function will just fail (actually, it will not compile).

If you are certain that your extension won't be called in a multithreaded state, you could make a global pointer

```
CPerl *pPerl;
```

and set it to the `pPerl` that was passed into your function, as in [Example 10.18](#). This works wonderfully until you try to use the extension with ActiveState's `PerlIS.dll` ISAPI application. This application gives quick Perl processing to Microsoft's IIS Web server. Because the extension works in a multithreaded environment, it is quite possible that your extension could be loaded in a multithreaded environment. Consider an example in which your Web server calls a CGI that uses your extension. If you set a global `CPerl` pointer, as in [Example 10.18](#), it is possible that while this script is running, another instance of it will be started. When your extension's function is called, the global is reset to the new `CPerl` object. If this occurs while your first script is accessing a function that relies on the global `pPerl`, you have a problem.

For more information on multithreading issues, see the section "Multithreading in an Extension" later in this chapter.

### **Example 10.18 Using a global `CPerl` pointer**

```

01. // Define the global CPerl object
02. CPerl *pPerl = NULL;

```

```

03.
04. void MyOtherFunction( SV* pSV );
05.
06. XS( MyFunction )
07. {
08.     dXSARGS;
09.     // Set the global CPerl object (::pPerl) with
10.    // the local CPerl object (pPerl).
11.    ::pPerl = pPerl;
12.    SV* pSV = ST( 0 );
13.    // Now call the other function
14.    MyOtherFunction( pSV );
15. }
16.
17. void MyOtherFunction( SV* pSV)
18. {
19.     // Now with the global CPerl object set all of the
20.     // Perl macros which rely on pPerl will simply use
21.     // the global pPerl.
22.     dXSARGS;
23.     // ...process pSV here...
24. }
```

One way around this is to prototype all your functions that will use Perl macros to accept a first parameter as a `CPerl` object pointer. I defined a couple of macros to do this. By using these macros, I can just redefine them if I use the core distribution's headers (because the core distribution does not recognize `CPerl`).

You can define the macros from [Example 10.19](#) in your header files. Whenever you need to create a function that makes any use of Perl macros, you can prototype it as follows:

```
void MyFunction( PERL_ONLY_ARG_PROTO );
void MyFunction2( PERL_ARG_PROTO char *pszString );
```

Notice that the first prototype takes only one parameter. Because of this, it uses the `PERL_ONLY_ARG_PROTO` macro. This macro defines that only one parameter is passed in, and it is a `CPerl` object. The second prototype takes more than one parameter, so it is using the `PERL_ARG_PROTO` macro.

Both prototypes return nothing, and they both accept a `CPerl` object if you compile them using the `ActiveState` headers; otherwise, the macros do not expand to anything when using the core distribution.

### ***Example 10.19 Establishing compatibility between the core distribution and ActiveState versions of Perl***

```
01. #if PERL_VERSION == 6 && ! defined( PERL_OBJECT )
02. #ifndef USE_THREADS
03.     #define PERL_ARG_PROTO          PerlInterpreter* my_perl,
04.     #define PERL_ONLY_PROTO         PerlInterpreter* my_perl
```

```

05.     #define PERL_ARG           my_perl,
06.     #define PERL_ARG_ONLY       my_perl
07. #else
08.     #define PERL_ARG_PROTO     pTHXo_
09.     #define PERL_ARG_ONLY_PROTO perl_thread *thr
10.     #define PERL_ARG           aTHXo_
11.     #define PERL_ARG_ONLY       aTHXo
12. #endif
13. #endif
14. #ifdef PERL_OBJECT
15.     #define PERL_ARG_PROTO     CPerl *pPerl,
16.     #define PERL_ONLY_ARG_PROTO CPerl *pPerl
17.     #define PERL_ARG           pPerl,
18.     #define PERL_ONLY_ARG       pPerl
19. #else // Must be using core distribution
20.     #define PERL_ARG_PROTO
21.     #define PERL_ONLY_ARG_PROTO
22.     #define PERL_ARG
23.     #define PERL_ONLY_ARG
24. #endif

```

In [Example 10.19](#), the `PERL_ONLY_ARG_PROTO` and `PERL_ARG_PROTO` macros are used and thus provide a full extension that could be compiled using either ActiveState or core distribution headers. The first block actually checks for Perl 5.006 when the `PERL_OBJECT` is *not* defined (this is the default ActiveState configuration for v5.006). If this state is true, we set the macros accordingly for Perl v5.006 to correctly compile.

## The Bootstrap Function

One of the things that Perl does after loading the extension is to run the bootstrap function. The DynaLoader loads the DLL and then looks for an exported function called `boot_ExtensionName`, where `ExtensionName` is the full name of the extension, replacing colon (:) characters with underscores (\_). Therefore, the `Win32::Test`'s bootstrap function will be called `boot_Win32__Test`.

When the bootstrap function is called, this is the extension's opportunity to make an association with a C function and a Perl subroutine. In other words, the bootstrap function will bind your C function to a Perl subroutine name so that Perl can make calls into your C code. This is done with the `newXS()` macro:

```
newXS( char *pszPerlName, XSSubroutine, char *pszFileName )
```

The first parameter (`pszPerlName`) is the full Perl function name. If our extension has a function name called `MyFunction()`, for example, this parameter would be `Win32::Test::MyFunction`.

The second parameter (`xSSubordinate`) is the name of the C function that was declared with the `XS()` macro. If the `MyFunction()` function was declared as `XS(MyFunction)`, this parameter would be `MyFunction` (notice there are no quotation marks around this).

The third parameter (`pszFileName`) is a filename string. This is used for debugging purposes and is typically declared as `_FILE_`, which is a compiler's macro for the currently compiling file.

You would have one `newXS()` macro for every function that is to be accessible from Perl.

The end of the bootstrap function needs to return a `TRUE` value back to the DynaLoader so that it knows the extension was successfully loaded. This is commonly performed by using the `XSRETURN_YES` macro. [Example 10.20](#) shows this.

### **Example 10.20 The bootstrap function**

```
01. XS( boot_Win32_Test )
02. {
03.     dXSARGS;
04.     char *pszFile = __FILE__;
05.
06.     newXS( "Win32::Test::MyFunction", MyFunction, pszFile );
07.
08.     XSRETURN_YES;
09. }
```

## **The .def File**

When creating a DLL file (as all Win32 Perl extensions are), you need to declare a definition (`.def`) file. This definition file will describe what the library is called and what functions to export. An in-depth explanation of the `.def` file is beyond the scope of this book. You might want to consult some Win32 programming manuals, however, such as *Win32 Programming* by Brent Rector and Joseph Newcomer (Addison Wesley Developers Press) and *Win32 Network Programming* by Ralph Davis (Addison Wesley Developers Press).

When creating a Perl extension, only two things really need to be included in the `.def` file: the `LIBRARY` and `EXPORTS` statements.

The `LIBRARY` statement declares the name of this particular DLL. Typically, this is the name of your extension, not the full namespace. If you are creating the `Win32::Test` extension, the library name is `Test`.

The `EXPORTS` statement just informs the linker what function is to be exported. Only one function really must be exported, the boot function. The name of the bootstrap function was described in the preceding section, "The Bootstrap Function."

After the `.def` file is created, it must be added to the C project so that when the extension is compiled and linked, the `.def` file will be included in the process. Have a look at [Example 10.28](#) for a sample of a `.def` file.

## **The `DllMain()` Function**

When a process loads a DLL, Windows will look for a function known as `DllMain()`. If the DLL has such a function, it is called. Likewise, when a process unloads a DLL (such as when the process is terminating), Windows again calls the `DllMain()` function.

## Note

All DLLs have a `DllMain()` function. If the programmer did not program one, the compiler created one that does nothing.

If a DLL has been loaded into a running program and a thread is created, once again, Windows will call the `DllMain()`, this time notifying it that a new thread has been created. Likewise, when a thread is terminated, Windows will call `DllMain()` to notify it of the terminating thread.

For many extensions, `DllMain()` is not an issue; most extensions appear to not make use of it. However, you might want to consider taking advantage of it for several reasons. These advantages are reviewed later during the detailed discussion of `DllMain()`.

To be more accurate, Windows does not call `DllMain()`; instead, it calls the C runtime library. The function that Windows really calls is `DllMainCRTStartup()`. This initializes the DLL's C runtime environment (declaring standard file handles, initializing static and global variables, and so on) and eventually calls the `DllMain()` function.

When the `DllMain()` function is called, it is passed in three parameters:

```
BOOL WINAPI DllMain( HINSTANCE hInstance, DWORD dwReason, LPVOID  
pvReserved );
```

The first parameter is the instance of the DLL. This is important because many Win32 API calls require the instance. The `DllMain()` function is your extension's chance to save the DLL's instance. You might want to save it to a global variable.

The second parameter is a `DWORD` that indicates the reason the function was called. Typically, your code will react differently depending on what the reason was for being called. [Table 10.5](#) lists the possible values for `dwReason`.

The third parameter is an `LPVOID`, which is reserved and should be ignored. It specifies further aspects of DLL initialization and cleanup.

**Table 10.5. Possible Values for `dwReason` in the `DllMain()` Function**

Value	Description
<code>DLL_PROCESS_ATTACH</code>	The DLL is being loaded for the first time by a process.
<code>DLL_THREAD_ATTACH</code>	The process just started a new thread.
<code>DLL_THREAD_DETACH</code>	The process just terminated a thread.
<code>DLL_PROCESS_DETACH</code>	The process is unloading the DLL.

The `DllMain()` function returns a Boolean value. If `TRUE` is returned, Windows will consider the function successful and continue processing as normal. If the function returns `FALSE`, Windows will report that the DLL was unable to be loaded. This means that your extension will have failed to load, and Perl will report this.

Basically, your `DllMain()` function should always return `TRUE` unless there is some compelling reason not to. If the DLL depends on another resource (such as a database or network connection) that is not available, for example, it might be prudent to return `FALSE` because the DLL will be unable to perform its job correctly.

[Example 10.21](#) demonstrates a very simple `DllMain()` function. In this example, line 9 stores the DLL's instance handle in `ghInstance`. This will be handy if later the extension needs to specify the instance—if it needs to load icon or dialog resources, for example. Line 15 increments a thread counter, `gdwThreads`. The only reason we do this is so that the extension knows how many threads are active at any given time.

### **Example 10.21 A simple `DllMain()` function**

```
01. HINSTANCE ghInstance = NULL;
02. DWORD gdwThreads = 0;
03. BOOL WINAPI DllMain( HINSTANCE hInstance, DWORD dwReason,
LPVOID lpReserved )
04. {
05.     BOOL bResult = TRUE;
06.     switch( dwReason )
07.     {
08.         case DLL_PROCESS_ATTACH:
09.             ghInstance = hInstance;
10.             // ...check for needed resources. If they are not
11.             // ...available set bResult = FALSE
12.             break;
13.         case DLL_THREAD_ATTACH:
14.             // Let's keep track of the number of threads-for kicks
15.             gdwThreads++;
16.             break;
17.         case DLL_THREAD_DETACH:
18.             gdwThreads--;
19.             break;
20.         case DLL_PROCESS_DETACH:
21.             break;
22.     }
23.     return( bResult );
24. }
```

## **Multithreading in an Extension**

Currently, threading is not fully supported by Perl. There is some experimental threading support, but it is just that: experimental. The `PerlIS.dll` extension, however, runs as an ISAPI application that is multithreaded. This means that any extension you write that might be used with `PerlIS.dll` needs to be multithread aware. For the most part, your code should not use global variables that

might need to change for every thread. Many extensions use a global `DWORD` to hold the last generated Win32 error, for example, something like `gdwLastError`.

If a Web server is using `PerlIS.dll` and it runs two scripts using the same extension at the same time, they might run into memory-contention issues. Suppose the first script generates an error and saves the error code into `gdwLastError`. Before it has a chance to report the error, however, the second script generates a different error. This new error's value is also stored in `gdwLastError`, essentially overwriting the first script's error code. When the first script reports its error, it will really be reporting the second script's error instead.

To make matters worse, extensions built using `PERL_OBJECT` (such as the ActiveState builds) require a special object called `pPerl` (which is a `CPerl` pointer or a `CPerlObj` pointer if you are using Perl 5.005 [and higher] with `PERL_OBJECT` support). All Perl-related functions are mapped as methods of this object. So a call to `croak()` is compiled as `pPerl->croak()`. For a function to call another function, it needs to somehow pass the `pPerl` object. Some extensions have done this by using a global `pPerl` pointer, as in [Example 10.22](#). This works just fine until you have two or more threads overwriting the `gpPerl` variable. This can cause some pretty funky crashes.

## Perl v5.006 and PERL\_OBJECT

*By default, Perl 5.006 (a.k.a. v5.6) does not define the `PERL_OBJECT` macro. Even if you download ActiveState's builds of 5.006, you will find that the macro was not defined when it compiled the code. This means that, unlike earlier versions of ActivePerl (such as v5.003), all your extensions should not be compiled with the `PERL_OBJECT` macro defined. Otherwise, binary compatibility will be broken, and the extensions will fail to function correctly.*

### **Example 10.22 A non-multithread-friendly way of handling `pPerl`**

```
01. CPerl *gpPerl;
02.
03. XS_MyFunction()
04. {
05.     // pPerl is automatically defined in XS functions
06.     gpPerl = pPerl;
07.     MyFunction2();
08. }
09.
10. void MyFunction2()
11. {
12.     CPerl *pPerl = gpPerl;
13.     // ...some code goes here...
14.     return;
15. }
```

There are several ways to handle multithreading issues, but the most practical advice is to avoid the use of global variables. Instead, either pass in a needed value (such as for `pPerl`) explicitly, as in [Example 10.23](#), or use thread local storage.

### **Example 10.23 A multithread-friendly way of handling pPerl**

```
01. XS(MyFunction)
02. {
03.     dXSARGS;
04.     // pPerl is automatically defined in XS functions
05.     MyFunction2( pPerl );
06. }
07.
08. void MyFunction2( Cperl *pPerl )
09. {
10.     // ...some code goes here that uses Perl macros so
11.     // ...it needs pPerl
12.     return;
13. }
```

### **Thread Local Storage**

Each thread in a process has its own stack but shares the same heap as the other threads. This means that automatic variables declared on the stack are not shared among threads, but memory allocated on the heap using functions such as `new()` and `malloc()` are easily accessible by other threads. Additionally, because global variables are stored in the heap, they too are accessible by all threads.

What would be ideal is for a thread to have its own set of data that is not accessible by other threads. Automatic variables are perfect for this, except they are destroyed when they fall out of scope, such as when a function returns. This brings us to the concept of *thread local storage* (TLS).

Win32 enables each thread to save information (in the form of a void pointer) that is unique to the thread. Whenever the thread needs that information, it can retrieve it. This is called thread local storage (or as I read it, local thread storage). Upon request, Win32 creates thread local storage *slots*, which are basically `void` pointers. Each slot is like an element in an array—there is an index number associated with it. If you request two TLS slots, Win32 will allocate two entries large enough to hold `void` pointers and return the index numbers to these slots.

Think of TLS as a banking system. Each customer (or thread) gets a savings account and a checking account. Each account (a TLS slot) has its own account number (or TLS slot index), such as 1 for savings and 2 for checking. When a customer goes to the bank to check her balances, she first provides her customer number (or thread ID) so that the teller knows which accounts to look at. Then she specifies which account number she intends to inquire about.

Just as the bank provides an account number for each account that a customer has, Win32 hands out a TLS slot index number for each variable that a thread needs to save. When the thread wants to retrieve data it stored in one of these slots, it queries the specified slot's TLS index number (just as the customer specifies the account number). Because the index number is shared by all threads, you can store the slot's index number in a global variable that all threads can access. This is akin to telling your DLL that all threads are to store some particular value (for example, the last generated error) in slot X.

What is really happening is that Win32 allocates an array of `void` pointers for each thread. When your code allocates a new TLS slot, it returns an index to one element in the array. Every time a thread stores or retrieves data using that TLS slot, Win32 just looks to the array that has been

allocated for that particular thread. The slot index number informs Win32 which element in the array to access.

Example 10.24 demonstrates the use of thread local storage. Detailed information regarding functions within the example follows.

**Example 10.24 Using thread local storage**

```
01. // Add Perl Headers here
02. // Create the global TLS index variables
03. DWORD gdwPerlIndex = 0;
04. DWORD gdwErrorIndex = 0;
05. HINSTANCE ghInstance = NULL;
06.
07. // Declare our function prototypes...
08. DWORD GetError();
09. void SetError( DWORD dwError );
10. CPerl *GetPerl();
11. void SetPerl( CPerl *pPerl );
12. void MyFunction();
13.
14. DWORD GetError()
15. {
16.     return( (DWORD) TlsGetValue( gdwErrorIndex ) );
17. }
18.
19. void SetError( DWORD dwError )
20. {
21.     TlsSetValue( gdwErrorIndex, (LPVOID) dwError );
22. }
23.
24. CPerl *GetPerl()
25. {
26.     return( (CPerl*) TlsGetValue( gdwPerlIndex ) );
27. }
28.
29. void SetPerl( CPerl *pPerl )
30. {
31.     TlsSetValue( gdwPerlIndex, (LPVOID) pPerl );
32. }
33.
34. XS(GetLastError)
35. {
36.     dXSARGS;
37.     DWORD dwError = GetError();
38.     ST( 0 ) = sv_newmortal();
39.     sv_setiv( ST( 0 ), (IV) dwError );
40.     XSRETURN( 1 );
41. }
42.
43. void MyFunction()
44. {
45.     CPerl *pPerl = GetPerl();
```

```

46.     dXSARGS;
47.     // ...now that you have a pPerl object
48.     // ...you can call Perl macros and such.
49.     return;
50. }
51.
52. BOOL WINAPI DllMain( HINSTANCE hInstance,
53.                       DWORD dwReason,
54.                       LPVOID lpReserved )
55. {
56.     BOOL bResult = TRUE;
57.     switch( dwReason )
58.     {
59.         case DLL_PROCESS_ATTACH:
60.             ghInstance = hInstance;
61.             // Allocate a TLS for the pPerl object...
62.             gdwPerlIndex = TlsAlloc();
63.             // Allocate a TLS for the last error...
64.             gdwErrorIndex = TlsAlloc();
65.             // ...check for needed resources. If they are not
66.             // ...available set bResult = FALSE
67.             if( FALSE == bResult )
68.             {
69.                 // If something went wrong and we are returning FALSE
70.                 // then we need to free up the TLS indexes!
71.                 TlsFree( gdwErrorIndex );
72.                 TlsFree( gdwPerlIndex );
73.             }
74.             break;
75.
76.         case DLL_THREAD_ATTACH:
77.             break;
78.
79.         case DLL_THREAD_DETACH:
80.             // If any memory has been allocated for this thread then
81.             // now is the time to free it!
82.             break;
83.
84.         case DLL_PROCESS_DETACH:
85.             TlsFree( gdwErrorIndex );
86.             TlsFree( gdwPerlIndex );
87.             break;
88.     }
89.     return( bResult );
90. }

```

To use TLS, your DLL must call `TlsAlloc()` for each value it intends to store when the process attaches to it. The `DWORD` returned by `TlsAlloc()` is the slot index number used by other TLS functions to identify the particular variable that has been stored. The prototype for `TlsAlloc()` is as follows:

```
DWORD TlsAlloc()
```

This function takes no parameters and creates a TLS slot. The slot's index number is returned as a `DWORD`.

If the `TlsAlloc()` function is successful, it allocates the TLS slot and returns the index number.

Example 10.24 stores two variables for every thread: a `CPerl` pointer and a `DWORD` representing the last error Win32 generated. The global indexes are defined in lines 3 and 4. Lines 62 and 64 call `TlsAlloc()` for each variable that will be stored. This allocates a TLS slot that can fit a `void` pointer.

After a TLS index has been created, your extension can store values in each slot by calling the `TlsSetValue()` function:

```
BOOL TlsSetValue( DWORD dwIndex, LPVOID pVoid )
```

The first parameter is a `DWORD`, which is the index value returned by `TlsAlloc()`. This index number refers to a TLS slot into which the value of the second parameter will be stored.

The second parameter is a value that has the size of a `void` pointer (typically 32 bits, but this can change from processor to processor). For most values that are not `void` pointers, you will need to type cast them to either (`LPVOID`) or (`VOID*`).

If the value was successfully set into the specified TLS index slot, the function returns `TRUE`; otherwise, it returns `FALSE` and no value is stored.

Example 10.24 creates two functions to set the Perl object and the error number: `SetPerl()` and `SetError()`. These functions set the value passed into them into their corresponding TLS slot.

After a value has been stored in a TLS slot, it can be retrieved by using `TlsGetValue()`:

```
LPVOID TlsGetValue( DWORD dwIndex )
```

The only parameter is a `DWORD`, which represents the TLS slot index. This is the same index used to store the value in a call to `TlsSetValue()` and is returned by a call to `TlsAlloc()`.

If the `TlsGetValue()` function succeeds, it returns the value stored in the specified TLS slot. That return value can be any value including zero, so there is no way to determine whether the function failed.

Example 10.24 also defines two functions to retrieve specific values from TLS storage: `GetPerl()` and `GetError()`.

After an application or DLL no longer requires TLS storage, a call to `TlsFree()` must be called for every TLS slot that was created:

```
BOOL TlsFree( DWORD dwIndex )
```

The only parameter is the index number of a TLS slot that is to be freed.

If the `TlsFree()` function succeeds, it returns `TRUE`; otherwise, it fails and returns `FALSE`. Really, the only time this function will fail is if the specified index has not already been allocated with a call to `TlsAlloc()`.

After the `TlsFree()` function has been called for a TLS slot index, that slot is no longer valid for *any* thread. Other slots that have been allocated are still usable; only those specified in a call to `TlsFree()` are no longer valid.

Also in [Example 10.24](#), lines 71 and 72 call `TlsFree()` for each TLS slot allocated in. Notice that this occurs when `DllMain()` has been called with `dwReason` set for `DLL_PROCESS_DETACH`. This is because Windows is telling the DLL that the process is about to unload it, and therefore it had better free up all resources that it had already allocated.

## A Practical Example

Now that most of the necessities have been covered, it is time to go about creating an example. I used Microsoft's Visual C++ 6.0 (actually MS Visual Studio), so you might need to make changes to your specific environment where necessary.

The first thing you need to do is create the files listed in [Examples 10.25](#) through 10.28:

- [Example 10.25](#) is a Perl script that demonstrates how to call the extension.
- [Example 10.26](#) is a Perl module required to load the extension.
- [Example 10.27](#) is the actual extension itself.
- [Example 10.28](#) is the DLL definition (`.def`) file.

The next step is to create a new Win32 DLL project (don't use any MFC classes). In MSVC 5.0 and 6.0, creating a new project automatically creates a new workspace for the project. Add the two files `test.cpp` ([Example 10.27](#)) and `test.def` ([Example 10.28](#)) to the project.

Now you need to define a couple of macros. You can hard code them into the `test.cpp` file, but that would limit the capability of others to use the file. You might want to define it in the project's compiler options. Set the options so that they define the `MSWIN32` macro.

```
#define MSWIN32
```

This will turn on Perl's Win32 options. Another macro needs to be defined only if you are doing any of the following:

- Using the ActiveState build
- Using build 5.005 and you want to use the `CPerlObj` class

This is necessary if your extensions will be running on Web servers using the ISAPI application `PerlIS.dll` or interacting with ActiveState's various builds. (Hint: All of ActiveState's builds use these classes.) If this is the case, define the macro as follows:

```
#define PERL_OBJECT
```

From this point, set the project's `include` path to include the Perl header files (obtained from [www.activestate.com](http://www.activestate.com), [www.perl.com](http://www.perl.com), or [www.cpan.org](http://www.cpan.org)).

If you are using the 5.004 core distribution, you will also need to inform the linker where the `perl.lib` file is so that it can link to the Perl library.

The next step is to compile and link!

Assuming that everything went well, you will have a resulting `test.dll` file (or whatever you named it). For the sake of this example, you need to make a directory for the `.dll` file. The directory name is the name of the extension. Because this extension is `Win32::Test`, the `.dll` file will be copied into a directory called `Win32\Test`. This path must be placed in the library's `AUTO` directory:

```
perl\lib\AUTO
```

Perl 5.005 (and higher), as well as the 5.004 core distribution, uses alternate directories for the Win32 extensions. The actual paths have changed from build to build, so your path might differ; however, the general method is to start with a site directory such as the following:

```
perl\site\lib\AUTO
```

Create the `Win32\Test` directory in the `AUTO` directory. Then copy the `test.dll` file into the directory. An example of the command line to copy the file into the appropriate directory is as follows:

```
xcopy test.dll c:\perl\site\lib\auto\win32\test\*.*
```

The `xcopy` command will create the directories if needed.

Next, you need to copy the Perl module into the correct module directory. The path is the same as where you copied the `.dll` file, except that it is not in the `AUTO` directory. An example of a command to copy the Perl module is as follows:

```
xcopy test.pm c:\perl\site\win32\*.*
```

Now you are ready to run the Perl script `test.pl`.

That is all there is to it!

### **Example 10.25 A Perl script (`test.pl`) using the `Win32::Test` extension**

```
01. # First load our new module which in turn will load
02. # our extension.
03. use Win32::Test;
```

```

04.
05. # Call the AddNumbers() function...
06. $Sum = Win32::Test::AddNumbers( 500, 42102 );
07.
08. # Now let's try the GetFileSizes() function...
09. if( $Count = Win32::Test::GetFileSizes( 'c:\temp\*.*',
$Data ) )
10. {
11.     # Unpack the array of data structures that was passed back
to
12.     # us from the extension's GetFileSizes() function. We must
13.     # use the unpack template of "A256l" because the extension's
14.     # FileStruct data structure consisted of:
15.     #
16.     #     char szName[256];
17.     #     long lSize;
18.     #
19.     # So we must unpack first the 256 characters followed by a
long
20.     # variable hence the unpack template of "A256l".
21.     # Note that we will use "A256l" x $$Count because we need to
unpack
22.     # for each file that was found. Since the extension
concatenated
23.     # all of the data structures together it is the same as one
long
24.     # array of structures.
25.     # The output is fed directly into a hash (%Files) because a
26.     # hash is an array with the format of
27.     # ( key, value, key2, value2, ... ). We will be unpacking the
data
28.     # in the format of ( file, size, file2, size2, ... ). This
makes
29.     # for a perfect hash with file names as the key and sizes as
value.
30.     %Files = unpack( "A256l" x $$Count, $Data );
31.     foreach $FileName ( keys( %Files ) )
32.     {
33.         print "The file '$FileName' is $$Files{$FileName}
bytes.\n";
34.     }
35. }
```

#### **Example 10.26 The Perl module (*test.pm*) for the *Win32::Test* extension**

```

01. # We must declare the name of this package (or module)
02. package Win32::Test;
03.
04. # Load the DynaLoader module so that
05. # we can dynamically load our extension
06. # If we need to export constant values and such
07. # we will need to use the Exporter but for this
```

```

08. # test we don't need it.
09. require DynaLoader;
10.
11. # We must add DynaLoader to the
12. # ISA array. This is ISA as in "is a" not as in the
13. # PC bus architecture (as opposed to PCI). The
14. # ISA array tells that we are inheriting functions
15. # and such from the specified module. We need to
16. # specify DynaLoader so that we can use its
17. # bootstrap function later.
18. @ISA= qw( DynaLoader );
19.
20. # Call the bootstrap function in our extension.
21. # This will register all the exported functions
22. # from our extension with Perl so that we can
23. # later call them.
24. bootstrap Win32::Test;
25.
26. # As silly as this last line looks it is imperative
27. # that it exists. By returning a 1 value we are reporting
28. # to Perl that the module has loaded alright
29. 1;

```

**Example 10.27 The Win32::Test extension (*test.c* or *test.cpp*, depending on whether you compile with C or C++)**

```

01. #ifdef __BORLANDC__
02.     typedef wchar_t wctype_t; /* in tchar.h, but unavailable
unless _UNICODE */
03. #endif
04. // WIN32_LEAN_AND_MEAN is a Microsoft Windows macro which
05. // is needed. This will prevent most of the Window specific
06. // stuff that we simply don't need
07. #define WIN32_LEAN_AND_MEAN
08. #include <windows.h>
09. // Gurusamy's right, Borland compilers need this here!
10. #include <stdio.h>
11. // Gurusamy's right, MS compilers need this here!
12. #include <math.h>
13. #if defined(__cplusplus) && !defined(PERL_OBJECT)
14.     extern "C" {
15. #endif
16.     #include "EXTERN.h"
17.     #include "perl.h"
18.     #include "Xsub.h"
19. #if defined(__cplusplus) && !defined(PERL_OBJECT)
20.     }
21. #endif
22.
23. // Declare some global variables
24. HINSTANCE ghInstance = NULL;
25. DWORD gdwThreads = 0;

```

```

26.
27. #if PERL_VERSION == 6 && ! defined( PERL_OBJECT )
28.   #ifndef USE_THREADS
29.     #define PERL_ARG_PROTO PerlInterpreter* my_perl,
30.     #define PERL_ARG_ONLY_PROTO PerlInterpreter* my_perl
31.     #define PERL_ARG my_perl,
32.     #define PERL_ARG_ONLY my_perl
33.   #else
34.     #define PERL_ARG_PROTO pTHXo_
35.     #define PERL_ARG_ONLY_PROTO perl_thread *thr
36.     #define PERL_ARG aTHXo_
37.     #define PERL_ARG_ONLY aTHXo
38.   #endif
39. #endif
40. #ifdef PERL_OBJECT
41.   // If you are using PERL_OBJECT with Perl 5.005/6 then
42.   // uncomment the next line. 5.005 changed the class from
43.   // CPerl to CPerlObj...
44.   // #define CPerl CPerlObj
45.   #define PERL_ARG_PROTO CPerl *pPerl,
46.   #define PERL_ONLY_ARG_PROTO CPerl *pPerl
47.   #define PERL_ARG pPerl,
48.   #define PERL_ONLY_ARG pPerl
49. #else
50.   // We get here if PERL_OBJECT is not defined such
51.   // as the core distribution of 5.004 and version
52.   // 5.005 without PERL_OBJECT support
53.   #define PERL_ARG_PROTO
54.   #define PERL_ONLY_ARG_PROTO
55.   #define PERL_ARG
56.   #define PERL_ONLY_ARG
57. #endif
58.
59. // Prototype our non Perl callable function
60. long ExtractLongFromSV( PERL_ARG_PROTO SV* pSV );
61.
62. XS( ExtensionAddNumbers )
63. {
64.   dXSARGS;
65.   long lNum1;
66.   long lNum2;
67.   long lSum;
68.   SV* pSV;
69.
70.   if( items != 2 )
71.   {
72.     // If 2 parameters were not passed in then complain about
73.     // it and print the syntax
74.     croak( "Usage: $Sum=Win32::Test::AddNumbers( $First,
75. $Second )" );
75.   }

```

```

76. // Get the first SV passed in and extract the value from it
77. pSV = ST( 0 );
78. lNum1 = ExtractLongFromSV( PERL_ARG pSV );
79.
80. // Get the second SV passed in and extract the value from it
81. pSV = ST( 1 );
82. lNum2 = ExtractLongFromSV( PERL_ARG pSV );
83.
84. lSum = lNum1 + lNum2;
85.
86. // Create a new SV with the sum
87. pSV = newSViv( lSum );
88.
89. // Tag the SV as mortal so that it will die after this
function
90. // has finished.
91. // Set the return value into the first position on the stack
92. ST( 0 ) = sv_2mortal( pSV );
93.
94. // Inform Perl that we have 1 value on the stack and let's
get
95. // back to our Perl script.
96. XSRETURN( 1 );
97. }
98.
99. // Our simple function to retrieve the IV (long) value from
an SV
100. long ExtractLongFromSV( PERL_ARG_PROTO SV* pSV )
101. {
102.     dXSARGS;
103.     long lValue = 0;
104.     // Check that the SV contains an integer and if so
105.     // extract it.
106.     if( SvIOK( pSV ) )
107.     {
108.         lValue = SvIV( pSV );
109.     }
110.     return( lValue );
111. }
112. // Define a structure to hold a file name and size
113. typedef struct
114. {
115.     char m_szPath[256];
116.     DWORD m_dwSize;
117. } FileStruct;
118.
119. XS( ExtensionGetFileSizes )
120. {
121.     dXSARGS;
122.     FileStruct File;
123.     WIN32_FIND_DATA fileFind;
124.     HANDLE hSearch;

```

```

125.     SV *pSV = newSVpv( "", 0 );
126.     char *pszPath = NULL;
127.     int iTotal = 0;
128.     BOOL bContinue = TRUE;
129.     unsigned int uiLength = 0;
130.     if( 2 != items )
131.     {
132.         croak( "Usage: $Count=Win32::Test::GetFileSizes( $Path,
133. $Data )" );
133.     }
134.     // Get the path to search. This is the first parameter
135.     // passed into
136.     // the function so it is the SV returned by ST(0)
137.     pszPath = SvPV( ST( 0 ), uiLength );
138.     // Now start searching the path using the Win32 file
139.     // search functions
140.     hSearch = FindFirstFile( pszPath, &fileFind );
141.     if( INVALID_HANDLE_VALUE != hSearch )
142.     {
143.         while( bContinue )
144.         {
145.             // Is the file that was found not a directory then
146.             // process it.
147.             if( !( fileFind.dwFileAttributes &
148. FILE_ATTRIBUTE_DIRECTORY ) )
149.             {
150.                 ZeroMemory( File.m_szPath, sizeof( File.m_szPath ) );
151.                 strncpy( File.m_szPath, fileFind.cFileName,
152. sizeof( File.m_szPath )
153.                     - 1 );
154.                 File.m_dwSize = fileFind.nFileSizeLow;
155.                 // Concatenate the data structure to an SV
156.                 sv_catpvn( pSV, (char*) &File, sizeof( File ) );
157.                 iTotal++;
158.             }
159.             bContinue = FindNextFile( hSearch, &fileFind );
160.         }
161.         FindClose( hSearch );
162.     }
163.     {
164.         pSV = &PL_sv_undef;
165.     }
166.     // Create a mortal SV with the integer value iTotal
167.     // then set it to the first element on the stack
168.     // (element 0).
169.     ST( 0 ) = sv_2mortal( newSViv( iTotal ) );
170.     // Take the SV passed in as the second parameter (which is
171.     // element 1 on the stack) and set its SV to point to
172.     // our pSV which contains all of the binary data structures

```

```

173. // we created for each of the files found. This does not
174. // do anything to the stack but instead sets the value of
175. // the SV which was passed into this function on the stack.
176. sv_setsv( ST( 1 ), pSV );
177. // Return to Perl informing the script that we are
returning
178. // with only one value (element 0 on the stack).
179. XSRETURN( 1 );
180. }
181.
182. // Now define the bootstrap. This is the only exported
183. // function. Your Perl module will call into this function
184. // which will register all XS() functions. Once this is done
185. // a Perl script (or your Perl module) can call any of the
186. // exported XS() functions directly.
187. // The name of this bootstrap is tied to the namespace of
your
188. // extension. In this case boot_Win32_Test represents the
189. // namespace Win32::Test
190. XS( boot_Win32_Test )
191. {
192.     dXSARGS;
193.     char *pszFile = __FILE__;
194.     newXS( "Win32::Test::GetFileSizes", ExtensionGetFileSizes,
pszFile );
195.     newXS( "Win32::Test::AddNumbers", ExtensionAddNumbers,
__FILE__ );
196.
197.     ST( 0 ) = &PL_sv_yes;
198.     XSRETURN( 1 );
199. }
200.
201. BOOL WINAPI DllMain( HINSTANCE hInstance,
202.                         dwReason,
203.                         lpReserved )
204. {
205.     BOOL bResult = TRUE;
206.     switch( dwReason )
207.     {
208.         case DLL_PROCESS_ATTACH:
209.             ghInstance = hInstance;
210.             break;
211.         case DLL_THREAD_ATTACH:
212.             // Let's keep track of the number of threads-for kicks
213.             gdwThreads++;
214.             break;
215.         case DLL_THREAD_DETACH:
216.             gdwThreads--;
217.             break;
218.         case DLL_PROCESS_DETACH:
219.             break;
220.     }

```

```
221.     return( bResult );
222. }
```

### **Example 10.28 The Win32::Test .def file (test.def)**

```
01. LIBRARY Test
02. EXPORTS
03.     boot_Win32__Test
```

## **Summary**

The folks who designed and wrote Perl should be congratulated for their work. They have designed a forward-thinking language that is totally extendable.

The capability to extend a language is not only a practicality but a must. Of course, no one language can include everything that is needed. To be sure, Perl does a fine job at doing almost anything a programmer needs, but when it comes to needing something it does not support, you can extend it rather easily.

You must consider several pitfalls when creating extensions, but with a bit of thought, none of them are insurmountable. Judging from the wide range of extensions available, it seems that this indeed is the case.

# **Chapter 11. Security**

Back in the old DOS and Windows 3.1 days, the way security was approached was actually pretty funny. The extent of it was that you could set a flag on a file indicating that it was read-only or that it was hidden, although there was nothing to prevent a user from using the `attrib.exe` command to change those settings. Because most users did not understand such flags, it was a reasonable way to secure data. This is still true with Win 95/98/ME machines.

However, when Windows NT came along, the concept of security took a bold step forward. NT defined a security system that was fairly foolproof. One would have to purposely hack into it to circumvent the intended security. With NT/Win2k/XP, security could be placed on anything: files, directories, printers, Registry keys, and the list goes on.

## **Security Concepts**

The way security is handled is not always obvious to a user. It is even less obvious to a programmer. Multiple concepts need to be understood before even the simplest of security can be applied. This section helps explain these concepts.

### **Security Subsystem**

Security is managed by the operating system by means of associating a list of permission flags with different user accounts. All these flag and account mappings are stored into a block of memory that represents the security of an object.

From a high level, you can describe Win32 security with groups of objects. Basically, you take a user account and associate it with various permissions and flags. This grouping of information is called an access control entry (ACE). You take a bunch of ACEs, group them together, and you have an access control list (ACL). If you take two different ACLs (a *discretionary* ACL and a *system* ACL) and group them together with an owner account name and a group account name, you have a *security descriptor* (SD).

It is the SD that you use to associate permissions with an object. All of the Windows API functions that create securable objects (such as files, Registry keys, network shares, and so on) accept an SD to indicate what permissions are to be linked to the given object.

## Securable Objects

Win32 consists of many different types of objects, such as files, printers, blocks of memory, and processes. Some of these objects can be secured; that is, they can have permissions applied to them. A file, directory, Registry key, named pipe, or network share is an example of a securable object.

To further complicate things, there are two different types of securable objects (actually this applies to nonsecurable objects as well): containers and noncontainers.

A *container* object is one that holds other objects. The most obvious example of this is a directory object. A directory can contain both files and other directories. Therefore, it is a container. As you can see, container objects can hold both container and noncontainer objects. However *non-container* objects cannot hold anything.

Examples of container objects include network shares, directories, printers, and Registry keys. Examples of noncontainer objects include files, Registry values, and print jobs.

## Using `Win32::Perms`

For the bulk of this chapter, we will be focusing on the `Win32::Perms` extension. Unlike the limited `Win32::FileSecurity` extension, `Win32::Perms` manages permissions in general. This enables you to do all sorts of fun things such as modifying permissions on various objects (Registry keys, files, directories, printers, network shares, and so on). It also provides various permission-related functions that are not available in other extensions.

One important aspect of the `Win32::Perms` extension, however, is that it attempts to look up accounts from their source. The source is typically a primary domain controller (PDC). However, this can be burdensome if you are working on a remote network or even offline.

You can instruct `Win32::Perms` to not look up the PDC each time it attempts to resolve an account or SID. This is done using the `LookupDC()` function:

```
$CurrentValue = Win32::Perms::LookupDC( [ $$newValue ] );
```

The optional parameter (`$newValue`) is either `TRUE (1)` or `FALSE (0)`. This value tells the extension whether or not to look up a domain controller when performing any SID and user account lookups. If this parameter is not passed in, the function will simply return the current setting.

The function will always return the current domain controller lookup setting. If a parameter was passed in, the setting is modified, and the return value indicates this new setting value.

## The Security Identifier

All security hinges on one central idea, that there is a user to whom you want to either grant or deny access. Generally, humans like to think of a user as a name such as "Dave" or "Administrator." This is fine for humans but inconvenient for computers. It is not efficient to always compare a string of characters as a name with a security permission. Likewise, human names can change. If a user account includes a last name as in "Jane\_Doe" and she remarries, it might change to "Jane\_Smith." This would require the Administrator to change all permissions that once pointed to "Jane\_Doe"; this can be quite time consuming and is error prone.

To solve this, Win32 introduced the concept of a security identifier (SID). This is a unique binary value that Win32 maps to a username. When you associate security permissions with a user, you are really mapping the permissions to the user's SID. This enables the username to change as often as needed because it does not change the SID. This enables a scenario such as user "Joe" leaving the company and being replaced by user "Jane." All the Administrator needs to do is rename Joe's account to "Jane" and change the password. Jane now has the exact same security access as Joe did.

A SID can be represented as either a binary or a text string. Generally speaking, most coders refer to the text version of a SID because it is easier for humans to interpret. Here's an example of such a text SID:

S-1-5-21-7893360589-1606980848-1708537768-500

or

S-1-5-21-7890445662-1935453667-1060284298-1003

This value identifies not only a particular user but also the domain in which the user account resides. Therefore, this value will not be the same for two accounts that are named the same but from different domains.

## ***Resolving User SIDs***

Two functions built into Win32 Perl provide mapping between SIDs and user accounts. These are the `Win32::LookupAccountSID()` and `Win32::LookupAccountName()` functions. Refer to [Chapter 9](#)'s section titled "Resolving User SIDs" for more details on these two functions.

The `Win32::Perms` extension also provides two convenient functions to map the SIDs to usernames. The first of these is the `Win32::Perms::ResolveSid()` function:

```
$TextSid = Win32::Perms::ResolveSid( $Account [, $BinarySid [, $Machine ] ]);
```

The first parameter (`$Account`) is the account you want to look up. This can be either a username (such as `Joe`) or a full account name including a domain (such as `Accounting\Joe`). You can

specify a machine's local account by specifying the machine name in the account, as in `\MyMachine\Administrator`. See [Table 11.1](#) for details on the different formats the account name can be.

The second, optional parameter (`$BinarySid`) will be set with the binary SID if the function is successful. Therefore, this *must* be a variable (no constants). Its previous value will be overwritten.

The optional third parameter (`$Machine`) specifies a particular machine to perform the SID lookup. This is useful when looking up a local account name such as "Administrator." The machine name must be prefixed with double backslashes as in `\Machine`.

If the function is successful, it returns a text-based SID. If the second parameter is passed in as a scalar variable, it will be set with the account's binary SID. If the function fails, nothing is returned.

[Example 11.1](#) shows how the `ResolveSid()` function works. The first few lines (2 through 4) discover the current user's username and domain. This information is passed into the function to represent the user's user account. Line 9 attempts to resolve the account to a SID. If it is successful, `$Sid{binary}` is set to the binary SID, and the function returns a text-based SID.

### **Example 11.1 Looking up a SID from an account name**

```
01. use Win32::Perms;
02. my $Domain = Win32::DomainName();
03. my $User = Win32::LoginName();
04. my $Account = sprintf( "%s%s", ( "" ne $$Domain )?
"$Domain\\\" : "", $User );
05. my %Sid = (
06.   text => "",
07.   binary => ""
08. );
09. if( $Sid{text} = Win32::Perms::ResolveSid( $Account,
$Sid{binary} ) )
10. {
11.   print "You are logged in as $Account.\n";
12.   print "Your SID is: $Sid{text}\n";
13.   print "Your binary SID is: '$Sid{binary}'\n";
14. }
15. else
16. {
17.   print "\tCouldn't lookup the SID for $Domain\\$User: $^E\n";
18. }
```

**Table 11.1. Different Win32::Perms Account Formats**

Account Format	Description
<code>UserName</code>	User account. The account is first searched for as a local account on the local machine. If it is not found, the default domain is examined.
<code>DomainName\UserName</code>	Domain account. The specified domain is searched for the account name.

**Table 11.1. Different Win32::Perms Account Formats**

Account Format	Description
\MachineName\UserName	Local machine account. The specified machine is searched for a local account. If the specified machine does not have such an account, the search continues on the default domain.

If you already have a SID but need to look up the user account associated with it, you can use the `ResolveAccount()` function:

```
$Account = Win32::Perms::ResolveAccount( $TextSid | $BinarySid [, $Machine ] );
```

The first parameter is the SID you are looking up. This can be either a text or binary SID. The function will determine which type of SID you are using and will handle it accordingly.

The optional second parameter indicates what machine on the network is to perform the lookup. If the SID represents a local account or group on a remote machine, you need to specify the remote machine in this parameter. The machine name must be prefixed with double backslashes, as in `\Machine`.

If the function is successful, it will return the account name in the form of "domain\user." Otherwise, the function returns nothing.

[Example 11.2](#) shows how both the `ResolveSid()` and `ResolveAccount()` functions work. Notice that line 9 looks up the current user's SID, and then line 15 uses that SID to look up the user's name. Line 15 passes in a binary SID, but it could just as well pass in the text SID (`$Sid{text}`) instead.

### **Example 11.2 Resolving accounts and SIDs**

```
01. use Win32::Perms;
02. my $Domain = Win32::DomainName();
03. my $User = Win32::LoginName();
04. my $Account = sprintf( "%s%s", ( "" ne $$Domain )?
"$Domain\\\" : "", $User );
05. my %Sid = (
06.   text => "",
07.   binary => ""
08. );
09. if( $Sid{text} = Win32::Perms::ResolveSid( $Account,
$Sid{binary} ) )
10. {
11.   print "You are logged in as $Account.\n";
12.   print "Your SID is: $Sid{text}\n";
13.   print "Your binary SID is: '$Sid{binary}'\n";
14.   print "Looking up your user account based on your SID:\n";
15.   if( my $LookupAccount =
Win32::Perms::ResolveAccount( $Sid{binary} ) )
```

```

16.  {
17.      print "\tBinary SID maps to: $LookupAccount\n";
18.  }
19. else
20. {
21.     print "\tCouldn't lookup the account: $^E\n";
22. }
23. if( my $LookupAccount =
Win32::Perms::ResolveAccount( $Sid{text} ) )
24. {
25.     print "\tText SID maps to: $LookupAccount\n";
26. }
27. else
28. {
29.     print "\tCouldn't lookup the account: $^E\n";
30. }
31. }
32. else
33. {
34.     print "\tCouldn't lookup the SID for $Domain\\$User: $^E\n";
35. }

```

## The Access Control Entry

For security to work, you have to associate a user with a particular permission. This is the job of the access control entry (ACE). An ACE is simply a memory structure that includes a user's SID, a *permission mask*, a *permission type*, and a *permission flag*. These four values together define the type of access a user has.

### **ACE Permission Bitmask**

The permission bitmask is a series of flags (actually they are just bits) that indicate the type of permission you are setting. For example, one flag might indicate read access, and another flag might indicate write access. It is difficult to definitively explain the permission mask because each securable object interprets permissions differently. For example, a file has a flag that indicates whether a user can execute it (provided the file is a program), but a directory does not have a concept of execution. The file system interprets the executable flag to indicate whether a user can open the directory. Now consider what this permission will mean to a printer or a named pipe or DCOM.

Each object is designed to interpret the permission bitmask in a way that applies to its use. This enables any programmer to create a program that uses Win32 permissions to determine how the program operates. You can imagine a Win32 service that only allows certain users to use the service and others to modify the service's settings. This is how the Win32 print manager enables you to set which users can print and manage jobs and manage the printer. See the "[Interpreting Permissions](#)" section later in this chapter for more details.

There are a few standard permission bitmask flags. Because a program can interpret each flag differently from another program, these standard flags provide a generic way for a coder to tell a program what to do. [Table 11.2](#) lists the different permissions that are applicable to all objects. These bitmask values can be OR'ed with each other to make a permission bitmask that will be

associated with a specific user. Other bitmasks flags can be set in addition to those in [Table 11.2](#), but they are object specific.

**Table 11.2. List of Standard ACE Permissions**

Permission Constant	Description
DELETE	The capability to remove or destroy the object.
READ_CONTROL	The capability to read control information from the object. This is needed to read what permissions are given to what accounts.
WRITE_DAC	The capability to write the discretionary access control list. This grants permission to modify permissions on the object.
WRITE_OWNER	The capability to edit the owner and group of the object.
SYNCHRONIZE	This is the right for the file to become signaled from a synchronization object, such as a mutex or semaphore, or a file notification.
STANDARD_RIGHTS_READ	Read permission.
STANDARD_RIGHTS_WRITE	Write permission.
STANDARD_RIGHTS_EXECUTE	Execute permission.
STANDARD_RIGHTS_ALL	Full permissions.
GENERIC_READ	Basic read access.
GENERIC_WRITE	Basic write access.
GENERIC_EXECUTE	Basic execute access.
GENERIC_ALL	All basic access rights (read, write, execute, delete).
FULL	Full access. This includes read, write, delete, modify, change owner, and so on.
CHANGE	This only grants the capability to modify the content of the object. This does not provide permissions to change permissions or owner.
READ	Read only.
WRITE	Write only.
NO_ACCESS	No access is provided.

## ACE Flags

There are two flavors of ACEs: effective and inherit-only ACEs. *Effective ACEs* are those that actually describe the type of permission applied to the object. An *inherit-only ACE* only exists so that an object can inherit it. For example, a directory object might have an inherit-only ACE that limits the write permission. Files that are added to the directory would inherit the ACE as an effective ACE. Therefore, the files would not limit who can write to them because the ACE is actually applied by the file. [Table 11.3](#) lists the various flags that an ACE can contain.

**Table 11.3. List of ACE Flags**

<b>Constant</b>	<b>Description</b>
CONTAINER_INHERIT_ACE	This allows child objects that are containers (such as subdirectories within a directory) to inherit the ACE. The inherited ACE is inheritable unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.
FAILED_ACCESS_ACE_FLAG	Used with system-audit ACEs in a SACL to generate audit messages for failed access attempts.
INHERIT_ONLY_ACE	Indicates an inherit-only ACE. The object that has this flag set does not use the ACE, but objects contained inside can inherit it. For example, a directory can have this flag, and only files within it can inherit the ACE and use it.
INHERITED_ACE	Windows 2000/XP: Indicates that the ACE was inherited. The system sets this bit when it propagates an inherited ACE to a child object.
NO_PROPAGATE_INHERIT_ACE	If the ACE is inherited by a child object, the system clears the OBJECT_INHERIT_ACE and CONTAINER_INHERIT_ACE flags in the inherited ACE. This prevents the ACE from being inherited by subsequent generations of objects.
OBJECT_INHERIT_ACE	Noncontainer child objects (such as Files) inherit the ACE as an effective ACE.
	For child objects that are containers (such as Directories), the ACE is inherited as an inherit-only ACE unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.
SUCCESSFUL_ACCESS_ACE_FLAG	Used with system-audit ACEs in a SACL to generate audit messages for successful access attempts.

## ACE Types

As if we have not already discussed enough about ACEs, there is one more aspect of an ACE to consider. This is the ACE *type*. ACEs come in several different types. Each of the types is defined in [Table 11.4](#). Every ACE has a type consisting of only one value from the table.

The ACE type is fairly important to understand. Win32 security is defined so that you can grant or deny access to a particular user. For example, let's say you have an ACE that associates the user "Joe" with full access to an object. If you specify the ACCESS\_DENIED\_ACE\_TYPE, that would mean "Joe" actually is denied full access; in other words, he has no access on the object.

Because an ACE can only be set to grant or deny access, it is common to use multiple ACEs for the same user. One ACE might specify all the permissions to which the user "Joe" has access, and another ACE would indicate what he is denied access to.

**Table 11.4. List of ACE Types**

ACE Type	ACE Description
ACCESS_ALLOWED_ACE_TYPE	The ACE defines permissions that grant a user access to the object.
ACCESS_DENIED_ACE_TYPE	The ACE defines permissions that prevent a user from accessing the object.
SYSTEM_AUDIT_ACE_TYPE	The ACE defines auditing flags.
SYSTEM_ALARM_ACE_TYPE	The ACE defines a Win32 alarm. This type of ACE has not yet been implemented into Windows NT/2000/XP.

### **The Access Control List (ACL)**

Permissions are not quite complete until you have a list of users whom you want to grant or deny access to some object. This means you would have a bunch of ACEs, one for each user and access. There are two different types of ACLs: *discretionary* and *system* ACLs.

The *discretionary ACL* (DACL) describes what we all have come to know as permissions. This is how can and cannot access an object.

The other type of ACL is the *system ACL* (SACL). This type of ACL is what administrators know as the audit list. It defines under what circumstances a specific user is audited. If a computer's auditing is turned on, any SACL that specifies that a given user is audited. An audit is tracked by adding an entry into the Win32 Event Log indicating the type of audit (failure or success). Not all objects support SACLs but most do.

An ACL does not really know what type it is. There is another structure that indicates the role of the ACL: the security descriptor.

### **The Security Descriptor (SD)**

So far we have discussed the SID, ACE, and ACL. Basically speaking, a SID is stored in an ACE. ACEs are stored in ACLs. For any given object, there are usually two ACLs (DACL and SACL). To make it all complete, we need something that groups all of this together, and that is the *security descriptor* (SD).

A security descriptor is a structure that groups a DACL and a SACL together. The SD is fairly simple; it only consists of four things:

- DACL
- SACL
- Owner SID
- Primary group SID

The two ACLs were discussed in the section called "[The Access Control List \(ACL\)](#)" earlier in this chapter, but the two SIDs need to be described.

Every securable object has an *owner*. This represents who created the object. You can see this when you look at permissions on a file or directory; the security properties dialog window displays the owner of the file or directory.

Regardless of what permissions have been set, the account represented by the owner SID *always* has full access to the object. If "Joe" has an ACE that denies him any access at all but he is the owner, he will have full access to the object.

Every securable object also has a *primary group*. This really does not do much. It only exists for POSIX compatibility. Unlike the owner, the group SID does not imply any special permissions.

Generally speaking, when an object is created, the object's owner SID is set to be the creating user, and the group is set to be the *default group* of the same user.

Security descriptors come in two different styles: *relative* and *absolute*. Each is very useful, but unfortunately, they are not necessarily interchangeable.

### ***Relative Security Descriptors***

A relative SD (sometimes referred to as a *self-relative security descriptor*) is one that consists of one block of memory. Each ACL and all its ACEs are stored in this block of memory. These types of SDs enable a program to serialize the SD to disk as a binary object. This is how Win32 stores SDs in the Registry.

### ***Absolute Security Descriptors***

An absolute SD is basically just a series of pointers to various ACE entries in each ACL. This type of SD is useful when a program manages several security descriptors. This enables several SDs to point to various different ACLs. The result is that you can conserve on memory by reusing different ACLs that are not from the same block of memory.

## **How Does Win32 Security Work?**

Any securable object can have an SD attached to it. It is up to the object to interpret the permissions. For example, the NTFS file system will examine a security descriptor on a given file and compare the permissions with the user account accessing the file. If the file system driver determines that the proper permissions exist for the user, permission is granted and the file system driver allows access. Otherwise, the NTFS driver will deny access and return an error to the calling application.

This is actually exciting news because you could write a Perl script that implements security much as NTFS does. Imagine you write an application that is only allowed to be run by select users. The script could store its security descriptor in a secure place (see the section "[Secure Storage](#)" later in this chapter), and every time the script is run, it could check this SD for runtime permissions. The Perl script itself would be the code that determines whether access is granted or denied.

### **Determining Access**

The general philosophy is that you are denied access unless you are granted access. This seems pretty obvious, but the more you ponder it, the less clear it becomes. A user can be granted or denied access. Obviously, if an ACE exists, it grants you access and then you can access the object.

If an ACE exists that denies you access, you are refused access to the object. However, what if there are no ACEs denying or granting you access? Having no ACE specifies that you are neither granted or denied access; therefore the Win32 OS assumes you do not have access.

Likewise, what if there are two ACEs, one granting and one denying access? Win32 will always store denied access ACEs in the beginning of the ACL. Therefore, when walking through the ACL to determine who has access, the denied ACEs are considered first. After a user has been explicitly granted or denied access, the OS stops examining the security descriptor. Therefore, any ACE denying access will occur first before any of the granting ACEs.

When a user attempts to access a secured object, all of the rights he is requesting (such as read, write, delete, and so on) have to be granted before he is given access to the object. If you try to open a file for reading and writing but you only have read access, you will be denied access to the file. If you then try to open the same file for reading only, you would be granted access.

## **Empty ACLs**

If a security descriptor contains a DACL that has no ACEs, the SD has an *empty DACL*. Such DACLs simply refuse any access to anyone. This is because of the "you are denied access unless you are granted access" philosophy. Empty SACLs indicate that no auditing is to occur.

## **NULL ACLs**

When there is no DACL or SACL specified in an SD, it is known to have a NULL DACL/SACL. This is because a NULL (zero value) is specified for the ACL pointer. In the case of a NULL DACL, Win32 assumes you want to specify that everyone has full access. This is what happens by default when Win32 converts a drive from the FAT format to the NTFS format. Win32 does not know which users to grant access to, so it does not create a DACL for each file. Therefore, each file and directory contains a NULL DACL that allows everyone full access.

Interestingly enough, if you use the Explorer program to show the permissions on a file with a NULL DACL, it will display "Everyone" as having full access. In this case, Win32 interprets the NULL DACL and displays the icon indicating that everyone has full access. If you were to add a permission where the user "Joe" is denied read access, Explorer would have to create a DACL and populate it with two ACEs: one for "Everyone" having full access and the other for "Joe" having read access denied.

A NULL SACL means that no auditing will occur. This is effectively the opposite behavior of a NULL DACL.

You can observe NULL DACLs by using the Explorer program on Windows 2000 or XP. Open a directory and select a file or directory. Right-click and hit the Properties context menu option. Select the Security tab. If there is only "Everyone" with full permissions, hit the Advanced button. A new dialog window will appear. In the Permissions tab, look to see whether there are any entries. If there are none, you have a NULL DACL. If the DACL was truly empty (that is, the DACL exists but there are no entries), there would be literally nobody with access in the Security tab of the first dialog window.

## ***Interpreting Permissions***

Any securable Win32 object can have a security descriptor attached to it. This means that access permissions can be applied to any object, thus enabling the operating system to be quite secure. However, it can be confusing for an administrator or programmer. After all, how can different objects that have nothing to do with one another relate to one set of permissions? The answer is that they cannot.

Win32 stores permissions as a series of bit flags (refer to the section "ACE Permission Bitmask" earlier in this chapter). Bit refers to a different permission. For example, one bit might indicate that a user has permission to write to a file, yet another bit might indicate that the user has permission to read from the file. All permissions are stored in ACE structures, and there are only 32 bits provided for these flags. This means that there are not enough bits for every application to have a bit dedicated to represent its type of permission. Therefore, bits must be reused.

Back in the section "How Does Win32 Security Work?", I described that it is up to an application to interpret permissions in a security descriptor. The `Win32::Perms` extension enables a script to transfer a security descriptor from one object to another, effectively copying permissions. However, because a file interprets permissions differently from a Registry key, it is important to note that transferring security descriptors across different object types can result in unintended security settings.

Consider a file with the directory read permission bit set (the `DIR_READ` permission). This permission sets bit 0 to `true (0x01)`. This happens to be the same for the network share read permission. Therefore, you could transfer that particular permission without incident. However, a directory has a `FILE_DELETE_CHILD` permission (allowing the user to remove a subdirectory) as well (`0x64`). There is no equivalent network share permission. Therefore, the two do not necessarily match.

Usually a program will interpret groups of particular permissions into a *friendly name*. This is a name that is a common way to refer to the grouping of permissions. For example, the Explorer program refers to `CHANGE` permissions as a friendly name for `READ`, `WRITE`, `EXECUTE` and a few other permissions. Because programs interpret permissions differently from one another, a friendly name such as `CHANGE` might consist of vastly different sets of permissions. See the "[Decoding Permissions](#)" section later in this chapter for more details on friendly names.

This also means that you might have valid permissions set on an object, but the program interpreting the permissions might not understand them. This is evident when setting permissions on a Windows NT 4.0 computer remotely from a Windows 2000 machine. In this case, the Win2k machine might set valid permissions that work just fine, but if you log on to the NT 4.0 machine and look at the permissions, you might get an error indicating that it does not recognize the permissions. This is a perfect example of how different programs interpret the same permission settings differently. The Windows 2000 Explorer program recognizes collections of permission bits differently than the NT 4.0 Explorer program does.

Even though permission settings differ from object to object, there are a few that do not. These are listed in [Table 11.3](#). All secure Win32 objects map these standard permissions to a valid permission setting. You can be assured that a user who has been granted the `STANDARD_RIGHTS_READ` permission, for example, will be given some type of read access to the object.

The common permission settings are listed in [Table 11.5](#) through [Table 11.9](#).

**Table 11.5. Permissions for a File**

Permission Constant	Description
ALL_ACCESS_FILE	Full control over the file.
FILE_APPEND_DATA	Append data to the end of a file.
FILE_EXECUTE	The capability to open a file. Think of this as allowing permission to run an application.
FILE_READ_ATTRIBUTES	The read attributes of a file (such as Read only, hidden, and so on).
FILE_READ_DATA	Read the contents of a file.
FILE_READ_EA	Read extended attributes from a file.
FILE_WRITE_ATTRIBUTES	Write attributes of a file (such as Read only, hidden, and so on).
FILE_WRITE_DATA	Write data to a file.
FILE_WRITE_EA	Write extended attributes to a file.

**Table 11.6. Permissions for a Directory**

Permission Constant	Description
ALL_ACCESS_DIR	Full control of the directory.
DIR_ADD_FILE	Add a file to a directory.
DIR_ADD_SUBDIR	Add a subdirectory to a directory
DIR_DEL_SUBDIR	The capability to delete a subdirectory.
DIR_EXECUTE	The capability to open a directory. Think of this as allowing permission to enter a directory.
DIR_TRAVERSE	allowing permission to enter a directory.
DIR_READ	Read the contents of a directory.
DIR_READ_ATTRIB	Read attributes of a directory (such as Readonly, hidden, and so on).
DIR_READ_EATTRIB	Read extended attributes from a directory.
DIR_WRITE_ATTRIB	Write attributes of a directory (such as Readonly, hidden, and so on).
DIR_WRITE_EATTRIB	Write extended attributes to a directory.

**Table 11.7. Permissions for Registry Keys**

Permission Constant	Description
KEY_ALL_ACCESS	Full control over the Registry key.
KEY_CREATE_LINK	Required to create a link to a Registry key.
KEY_CREATE_SUB_KEY	Required to create a subkey in a Registry key.

**Table 11.7. Permissions for Registry Keys**

Permission Constant	Description
KEY_ENUMERATE_SUB_KEYS	Enumerate subkeys in a Registry key.
KEY_NOTIFY	Required to request change notifications for a Registry key or its subkeys.
KEY_QUERY_VALUE	Read a value in a Registry key.
KEY_READ	Open a Registry key.
KEY_EXECUTE	
KEY_WRITE	Write a value to a Registry key.
KEY_SET_VALUE	

**Table 11.8. Permissions for a Shared Network Directory**

Permission Constant	Description
FULL_SHARE_CONTROL	Full control over the share.
FILE_SHARE_READ	Read content through the network share.
FILE_SHARE_WRITE	Write content through the network share.
FILE_SHARE_DELETE	Delete content through the network share.

**Table 11.9. Permissions for Printers**

Permission Constant	Description
PRINTER_ALL_ACCESS	Full access over the printer.
PRINTER_READ	Read status of a printer.
PRINTER_WRITE	Write to a printer (basic printing).
PRINTER_EXECUTE	Open a printer for interaction.
JOB_ALL_ACCESS	Full control over print jobs.
JOB_READ	Read a print job.
JOB_WRITE	Write (create) a new print job.
JOB_EXECUTE	Execute (print) a print job.

## Accessing Win32 Security Using *Win32::Perms*

The `Win32::Perms` extension provides access to Win32 security. To use this extension, you will have to load the extension by specifying the following line (preferably at the beginning of your script):

```
use Win32::Perms;
```

### **Domain Controller Lookups**

If you are using `Win32::Perms`, you might want to add the following line after you load the extension:

```
Win32::Perms::LookupDC( 0 );
```

This will prevent `Win32::Perms` from automatically looking up a domain controller for each lookup. Over slow networks or when you are disconnected from the net, this function can save considerable time.

### **Creating Security Descriptors**

Before you can do anything with permissions, you need to first procure a security descriptor (refer to the earlier section "[The Security Descriptor \(SD\)](#)"). When you have an SD, you can then add and remove permissions as much as you want.

A `Win32::Perms` object is a Perl representation of a security descriptor. You can create one with the `new()` function:

```
$PermsObj = new Win32::Perms( [ $$Path | $PermsObject [, $Type ] ] );
```

The optional first parameter (`$Path` or `$PermsObject`) is either a string representing a securable object's path or it is a valid `Win32::Perms` object. If a string is specified, it must follow one of the path formats listed in [Table 11.10](#). Additionally, blessed Perl objects can be passed into this function. For example, a `Win32::Registry` object can be passed in. This would cause the `Win32::Perms` object to reflect the permissions of the Registry key.

The optional second parameter (`$Type`) forces the type of securable object that the security descriptor represents. This is very handy if you specify a path of "" as if you want to create an empty SD, but it will be used for a particular type of securable object. See [Table 11.11](#) for a list of valid types.

If successful, the function will return a `Win32::Perms` object. If a path was specified, the object will represent the security descriptor of the specified path.

**Table 11.10. Different *Win32::Perms* Path Formats**

<b>Path Format</b>	<b>Example</b>
Normal path	c:\temp\file.txt d:\Program Files\Office \\server\share \\server\share\dir\file.txt \\server\printer HKEY_LOCAL_MACHINE\Software \\machine\HKEY_LOCAL_MACHINE\Software
Object type path	(file: and dir: are synonymous) file:c:\temp\file.txt dir:d:\Program Files\Office share:\\server\share file:\\server\share\dir\file.txt printer:\\server\printer registry:HKEY_LOCAL_MACHINE\Software registry:\\machine\HKEY_LOCAL_MACHINE\Software

**Table 11.11. List of Win32::Perms Types**

<b>Constant</b>	<b>Description</b>
PERM_TYPE_NULL	Creates a NULL descriptor. This is the same as a
PERM_TYPE_UNKNOWN	security descriptor with a NULL DACL/SACL. If an object of this type is used to set permissions, they will be set to the Default state, which is FULL access to Everyone.
PERM_TYPE_NONE	
PERM_TYPE_REGISTRY	The path is a Registry key.
PERM_TYPE_File	The path is to a file.
PERM_TYPE_SHARE	The path is to a network share.
PERM_TYPE_PRINTER	The path is to a network printer.
PERM_TYPE_SD	The object created is only an SD with no association with any objects.
PERM_TYPE_SERVICE	The object represents a Win32 service object. This type is not implemented.
PERM_TYPE_KERNEL	The object represents a securable Win32 kernel-level object. This type is not implemented.
PERM_TYPE_PRIVATE	The object represents a securable private object that a non-operating system component has created. This type is not implemented.
PERM_TYPE_DIRECTORY	The path is to a directory.

### Tip:

When specifying a path for a Registry key, you can shorten the path by using the following abbreviations:

HKCR	HKEY_CLASSES_ROOT
HKCC	HKEY_CURRENT_CONFIG
HKCU	HKEY_CURRENT_USER

HKCR	<i>HKEY_CLASSES_ROOT</i>
HKLM	<i>HKEY_LOCAL_MACHINE</i>
HKU	<i>HKEY_USERS</i>

For example, if you wanted to access the *HKEY\_LOCAL\_MACHINE\Software* key, you could simply specify a path of *HKLM\Software*. This works on remote machines as well as in *\MyMachine\HKLM\Software*.

[Example 11.3](#) shows how you can pass in a *Win32::Registry* object into the *new()* function. The *Win32::Perms* extension maps the security permissions to the Registry key object that the *\$Key* variable represents (line 5).

### **Example 11.3 Passing in a Win32::Registry object**

```

01. use Win32::Perms;
02. use Win32::Registry;
03. my $Key;
04. $HKEY_LOCAL_MACHINE->Open( "Software", $Key ) || die;
05. my $PermsObj = new Win32::Perms( $Key ) || die;
06. my @List;
07. if( $PermsObj->Get( \@List ) )
08. {
09.     foreach my $Ace ( @List )
10.    {
11.        my @Perms;
12.        Win32::Perms::DecodeMask( $Ace, \@Perms );
13.        print " Account:: $Ace->{Account}\n";
14.        print " Permissions::\n";
15.        foreach my $Perm ( @Perms )
16.        {
17.            print " $Perm\n";
18.        }
19.        print "\n";
20.    }
21. }
22. $PermsObj->Close();
23. $Key->Close();

```

### **Closing a Win32::Perms Object**

Once a *Win32::Perms* object is no longer used, you can force it to destroy itself by calling the *Close()* method:

```
$PermsObj->Close();
```

This will destroy the `Win32::Perms` object and thus free up any used memory. After destroyed, the object is no longer valid and cannot be accessed.

Refer to line 22 in [Example 11.3](#) for an example of closing a `Win32::Perms` object. Note that it is not imperative that you close the object. After the object falls out of scope, it will automatically be closed.

## Accessing the Type Of SD

When you have a `Win32::Perms` object, it can be useful to query what type of object it represents. This is done with the `GetType()` method:

```
$ObjectType = $PermsObj->GetType();
```

The method will return a value that represents one of the `Win32::Perms` types listed in [Table 11.12](#).

[Example 11.4](#) demonstrates the use of the `GetType()` method. Lines 3 through 10 create a hash consisting of keys that represent the various `Win32::Perms` object types. Line 12 prints out the object type by first calling the `GetType()` method and then printing the hash value for the object type key.

### **Example 11.4 Accessing the type of Win32::Perms object**

```
01. use Win32::Perms;
02. my $Path = shift @ARGV || $0;
03. my %TYPE = (
04.   eval( PERM_TYPE_NULL ) => 'NULL SD',
05.   eval( PERM_TYPE_NONE ) => 'None',
06.   eval( PERM_TYPE_UNKNOWN ) => 'Unknown type',
07.   eval( PERM_TYPE_FILE ) => 'File',
08.   eval( PERM_TYPE_SHARE ) => 'Network share',
09.   eval( PERM_TYPE_SD ) => 'Generic SD',
10.  eval( PERM_TYPE_PRINTER ) => 'Networked printer',
11.  eval( PERM_TYPE_REGISTRY ) => 'Registry key',
12.  eval( PERM_TYPE_SERVICE ) => 'Service',
13.  eval( PERM_TYPE_KERNEL ) => 'Kernel',
14.  eval( PERM_TYPE_PRIVATE ) => 'Private',
15.  eval( PERM_TYPE_DIRECTORY ) => 'Directory'
16. );
17. my $PermsObj = new Win32::Perms( $Path ) || die;
18. print "The path represents a ", $TYPE{$PermsObj->GetType()}, ".\n";
```

## Accessing an SD's Path

Usually there is a path associated with a `Win32::Perms` object. You can query and change this path with the `Path()` method:

```
$Path = $PermsObj->Path( [$NewPath] );
```

The optional parameter (`$NewPath`) is the path to a new path. The path can only be a "normal path," as described in [Table 11.10](#). Likewise, it *must* be for the same format as the object it represents. In other words, if you create a `Win32::Perms` object for a file, setting the path to point to a Registry key is not valid. It is undefined what would happen in this case.

[Example 11.5](#) illustrates how to query and change the path of a `Win32::Perms` object.

### ***Example 11.5 Changing the path***

```
01. use Win32::Perms;
02. my $Path = shift @ARGV || $0;
03. my $PermsObj = new Win32::Perms( $Path ) || die;
04. print " Path:: ", $PermsObj->Path(), "\n";
05. $PermsObj->Path( "c:\\my\\path" );
06. print " Modified path:: ", $PermsObj->Path(), "\n";
```

## **Accessing an SD's Owner and Primary Group**

Each SD has an owner and a primary group. You can query and set them using the `Owner()` and `Group()` methods:

```
$Owner = $PermsObj->Owner( [ $NewOwner ] );
$Group = $PermsObj->Group( [ $NewGroup ] );
```

The only parameters (`$NewOwner` and `$NewGroup`) are optional and represent an account or group name. If this parameter is specified, an attempt to set the owner or group with the specified account/group is made. This parameter must be an account string; it cannot be a SID. Refer to [Table 11.1](#) for a list of valid account formats.

The two methods return the current owner or primary group name associated with the security descriptor. If an account is passed in, the returned string represents the new account. [Example 11.6](#) shows how to use the `Owner()` and `Group()` methods.

### ***Example 11.6 Querying an object's owner and primary group***

```
01. use Win32::Perms;
02. my $Path = shift @ARGV || $0;
03. my $PermsObj = new Win32::Perms( $Path ) || die;
04. print " Path:: ", $PermsObj->Path(), "\n";
05. print " Owner:: ", $PermsObj->Owner(), "\n";
06. print " Group:: ", $PermsObj->Group(), "\n";
```

## **Listing Who Has Access**

Because a `Win32::Perms` object represents a security descriptor, it can contain a DACL and SACL. You can query the actual ACEs within the DACL and SACL. This is done by calling the `Get()` method:

```
$PermsObj->Get( [ \@ACEList ] );
```

The only parameter (`\@AceList`) is optional. It must be a reference to an array. The array will be cleared and then populated with a series of anonymous hashes. Each hash represents either an ACE or an owner or primary group SID.

Each hash entry contains the following keys:

- **Access:** This string is one of the values listed in the following table:

<b>Access Value</b>	<b>Description</b>
Allowed	This is equivalent to the ACCESS_ALLOWED_ACE_TYPE ace type.
Denied	This is equivalent to the ACCESS_DENIED_ACE_TYPE ace type.
Audit	This is equivalent to the SYSTEM_AUDIT_ACE_TYPE ace type.
Alarm	This represents a system alarm ACE type. Win32 does not yet support this type of ACE.
Unknown	Perl was unable to determine the type of ACE. It probably is invalid.

- **Account:** This is the name of the user account to which the ACE refers.
- **Domain:** This is the domain where the account resides.
- **EnTRY:** This indicates to what ACL the ACE belongs. This value is either `DACL` or `SACL`.
- **Flag:** This is the ACE's flags. This is a combination of the flags listed in [Table 11.3](#).
- **Mask:** This is the permission bitmask in numeric format. Some of the flags might come from [Table 11.2](#).
- **ObjectName:** This is a text string that identifies the type of object. This might be `File`, `Directory`, `Registry`, and so on.
- **ObjectType:** This is the numeric type that identifies the type of object. The list of valid object type values is found in [Table 11.12](#).
- **SID:** This is the text representation of the user's SID.
- **Type:** This is the ACE's type value. This can be any value from [Table 11.4](#).

The method returns the number of ACEs it enumerated. This includes ACEs from the DACL and the SACL.

## Tip

*Early in development of the `Win32::Perms` extension, there was no `Get()` method. Coders relied on a debug method called `Dump()`. Both methods perform similar functions, but the `Get()` method is the preferred technique to retrieve permission information from a `Win32::Perms` object.*

[Example 11.7](#) uses the `Get()` method to list the various ACEs on a given object.

### **Example 11.7 Displaying ACE information**

```
01. use Win32::Perms;
02. my @List;
```

```

03. my $Path = shift @ARGV || $0;
04. my $PermsObj = new Win32::Perms( $Path ) || die;
05. if( my $iTotal = $PermsObj->Get( \@List ) )
06. {
07.     print "Total of $iTotal ACES:\n";
08.     foreach my $Ace ( @List )
09.     {
10.         # Let's only look at permissions (not audits)
11.         next unless( "DACL" eq $Ace->{Entry} );
12.         my $Account = ( "" ne $Ace->{Domain} )? "$Ace-
>{Domain}\\" : "";
13.         $Account .= $Ace->{Account};
14.         print "\tAccount: $Account\n";
15.         print "\tPath: ", $PermsObj->Path(), "\n";
16.         print "\tType: ", $Ace->{ObjectName}, "\n";
17.         print "\tAccess: ", $Ace->{Access}, "\n";
18.         print "\n";
19.     }
20. }

```

## Decoding Permissions

Using the `Get()` method (refer to the section called "Listing Who Has Access"), you can discover what user has what permissions. However, the permission bitmask is just a numeric value; it does not make much sense to humans. This is why the `DecodeMask()` function was created. This function disassembles the bitmask into a human readable form:

```
$Count = Win32::Perms::DecodeMask( $BitMask | $HashRef [, \@Perms
[, @FriendlyPerms ] ] );
```

The first parameter (`$BitMask` or `$HashRef`) is either the numeric bitmask value or a hash reference obtained with a call to the `Get()` method. It is preferred to pass in the hash reference because it contains information regarding how to interpret the permissions. Passing in a bitmask value alone will only result in generic permissions.

The optional second parameter (`\@Perms`) is a reference to an array. This array will be populated with a list of permission strings. Each string represents a particular permission that the passed-in bitmask or hash reference has set. This list is the entire list of permissions.

The optional third parameter (`\@FriendlyPerms`) is a reference to an array. This array will be populated with a list of "friendly" permissions. These are permission strings that relate to the particular type of object. In many cases, there are no "friendly" translations of the regular permissions. Therefore, it is not uncommon to have no "friendly" permissions but many regular permissions for a given ACE.

Because of how often this function is used, the extension exports the function's name into the main namespace. This means you can call the function from a script in one of two ways:

```
Win32::Perms::DecodeMask();
DecodeMask();
```

The latter example is convenient because you don't have to type in the full namespace of the function's extension.

Example 11.8 uses both the `Get()` and `DecodeMask()` methods. Line 5 procures the list of ACEs, and lines 8 through 33 walk through each ACE to display its various properties. Line 14 decodes the permission mask into normal and "friendly" permission strings. Then lines 21 through 24 and 26 through 30 display the normal and friendly permission strings, respectively.

### ***Example 11.8 Displaying ACE information with permissions***

```
01. use Win32::Perms;
02. my @List;
03. my $Path = shift @ARGV || $0;
04. my $PermsObj = new Win32::Perms( $Path ) || die;
05. if( my $iTotal = $PermsObj->Get( \@List ) )
06. {
07.     print "Total of $iTotal ACEs:\n";
08.     foreach my $Ace ( @List )
09.     {
10.         # Let's only look at permissions (not audits)
11.         next unless( "DACL" eq $$Ace->{Entry} );
12.         my( @Perms, @FriendlyPerms );
13.         my $Account = ( "" ne $$Ace->{Domain} )? "$Ace-
>{Domain}\\" : "";
14.         Win32::Perms::DecodeMask( $Ace, \@Perms, \@FriendlyPerms );
15.         $Account .= $Ace->{Account};
16.         print "\tAccount: $Account\n";
17.         print "\tPath: ", $PermsObj->Path(), "\n";
18.         print "\tType: ", $Ace->{ObjectName}, "\n";
19.         print "\tAccess: ", $Ace->{Access}, "\n";
20.         print "\tPermissions:\n";
21.         foreach my $Perm ( @Perms )
22.         {
23.             print "\t\t$Perm\n";
24.         }
25.         print "\tFriendly Permissions:\n";
26.         foreach my $Perm ( @FriendlyPerms )
27.         {
28.             print "\t\t$Perm\n";
29.         }
30.         print "\n";
31.     }
32. }
```

A better example of how you can use the `Get()` and `DecodeMask()` methods is by looking at Example 11.9. This particular script originally appeared in my second book, *Win32 Perl Scripting: The Administrator's Handbook*, (New Riders Publishing), and because it illustrates the use of

`DecodeMask()`, I am reprinting it here. This script will examine the permission bitmask for a given object and display permissions on the screen using characters to symbolize each permission.

The permission characters that can be displayed from this script are as follows:

R	Read access
W	Write access
X	Execute access
D	Delete access
P	Modify permissions access
O	Modify owner access
A	Full access

If a character is capitalized, the particular permission is granted to the specified user. If the character is lowercase, the permission is explicitly denied to the user. If the character is not displayed, the permission was neither granted nor denied; this results in the user not being given that particular permission.

Lines 50 through 64 check to see what file paths were passed in on the command line and then generates a list of files by expanding any wildcards. If no filenames or paths were passed in, all of the files in the current directory will be examined.

Line 70 creates a new `Win32::Perms` object. Line 83 calls the `Get()` method on the freshly created object. The result is an array (`@List`) of ACEs. Lines 84 through 160 then generate the permissions string for each ACE. In line 110, the `DecodeMask()` function is called to get lists of permissions. Notice that the script does not specify the function's full namespace (`Win32::Perms`), as [Example 11.9](#) does. There is no significance to this other than to demonstrate that the function's name is exported into the main namespace.

### ***Example 11.9 Displaying who has permissions on specified objects***

```
01. use Win32::Perms;
02. # The %PERM hash maps a permission type to its position in the
03. # permission string array. Notice that there is no 0 position
04. # since that is reserved for unhandled permission types (and
05. # we won't print it).
06. %PERM = (
07.   '' => 0,
08.   R => 1,
09.   W => 2,
10.   X => 3,
11.   D => 4,
12.   P => 5,
13.   O => 6,
14.   A => 7,
15. );
16.
```

```

17. %MAP = (
18.     'FILE_READ_DATA'      => 'R',
19.     'GENERIC_READ'        => 'R',
20.     'KEY_READ'            => 'R',
21.     'DIR_READ'            => 'R',
22.
23.     'FILE_WRITE_DATA'    => 'W',
24.     'KEY_WRITE'           => 'W',
25.     'GENERIC_WRITE'       => 'W',
26.     'FILE_APPEND_DATA'   => 'W',
27.     'DIR_ADD_SUBDIR'     => 'W',
28.     'DIR_ADD_FILE'        => 'W',
29.
30.     'DELETE'              => 'D',
31.     'FILE_DELETE_CHILD'  => 'D',
32.
33.     'FILE_EXECUTE'        => 'X',
34.     'FILE_TRAVERSE'       => 'X',
35.     'GENERIC_EXECUTE'     => 'X',
36.     'DIR_TRAVERSE'        => 'X',
37.     'DIR_EXECUTE'         => 'X',
38.
39.     'CHANGE_PERMISSION'   => 'P',
40.
41.     'TAKE_OWNERSHIP'      => 'O',
42.
43.     'FILE_ALL_ACCESS'     => 'A',
44.     'GENERIC_ALL'          => 'A',
45.     'DIR_ALL_ACCESS'       => 'A',
46.     'STANDARD_RIGHTS_ALL' => 'A',
47.     ''                     => '',
48. );
49.
50. push( @ARGV, "*.*" ) unless( scalar @ARGV );
51. foreach my $Mask ( @ARGV )
52. {
53.     my( @List ) = glob( $Mask );
54.     if( ! scalar @List )
55.     {
56.         push( @List, $Mask );
57.     }
58.     foreach my $Path ( @List )
59.     {
60.         print "\nPermissions for '$Path':\n";
61.         ReportPerms( $Path );
62.         print "\n\n";
63.     }
64. }
65.
66. sub ReportPerms
67. {
68.     my( $Path ) = @_;

```

```

69. my( $Acct, @List );
70. my( $Perm ) = new Win32::Perms( $Path );
71. my( %PermList ) = ();
72. my( $MaxAcctLength ) = 1;
73.
74. if( ! $Perm )
75. {
76.     print "Can not obtain permissions for '$Path'\n";
77.     return;
78. }
79.
80. printf( " Owner:: %s\n Group: %s\n",
81.         $Perm->Owner(),
82.         $Perm->Group() );
83. $Perm->Get( \@List );
84. foreach my $Acct ( @List )
85. {
86.     next unless( defined $Acct->{Access} );
87.     my $PermMask = 0;
88.     my( $Mask, @M, @F );
89.     my( $DaclType );
90.     my $bAllowAccess = ( "Deny" ne $$Acct->{Access} );
91.     my $String;
92.     my $Account;
93.
94.     next if( $Acct->{Entry} ne "DACL" );
95.
96.     if( "" eq $$Acct->{Account} )
97.     {
98.         $Account = $Acct->{SID};
99.     }
100.    else
101.    {
102.        $Account = "$Acct->{Domain}\\" if( "" ne $$Acct-
103. >{Domain} );
104.        $Account .= $Acct->{Account};
105.    }
106.    if( length( $Account ) > $MaxAcctLength )
107.    {
108.        $MaxAcctLength = length( $Account )
109.    }
110.    $iTotal++;
111.    DecodeMask( $Acct, \@M, \@F );
112.    foreach my $Mask ( @M )
113.    {
114.        next unless( defined $MAP{$Mask} );
115.        my $Mapping = $MAP{$Mask};
116.        my $Permission = $PERM{$Mapping};
117.        $PermMask |= 2**$Permission;
118.    }
119.    $DaclType = $Acct->{ObjectName};
120.    if( 2 == $Acct->{ObjectType} )

```

```

120.      {
121.          # We have either a file or directory. Therefore we need
122.          # figure out if this DACL represents an object (file)
123.          # or
124.          # a container (dir)...
125.          if( $Acct->{Flag} & DIR )
126.          {
127.              $DaclType = "Directory";
128.          }
129.          else
130.          {
131.              $DaclType = "File";
132.          }
133.          if( ! defined $PermList{$Account}->{$DaclType} )
134.          {
135.              # Create the permission string array. The first element
136.              # in the array must be blank since all unhandled permissions
137.              # will default
138.              # to that position (and we won't print it).
139.              my $TempHash = [
140.                  " ",
141.                  scalar( keys( %PERM ) ) );
142.              $PermList{$Account}->{$DaclType} = $TempHash;
143.
144.          }
145.          foreach $Mask ( keys( %PERM ) )
146.          {
147.              if( $PermMask & 2**$PERM{$Mask} )
148.              {
149.                  $String = $PermList{$Account}->{$DaclType};
150.                  # If we already have a denied permission then skip
151.                  # this step
152.                  # since denied access overrides any explicitly
153.                  # allowed access
154.                  if( $String->[$PERM{$Mask}] !~ /[a-z]/ )
155.                  {
156.                      my $TempMask = $Mask;
157.                      $TempMask = lc $Mask if( 0 == $bAllowAccess );
158.                      $String->[$PERM{$Mask}] = $TempMask ;
159.                  }
160.              }
161.          }
162.          if( ! $iTotal )
163.          {
164.              # There are no DACL entries therefore...

```

```

165.     print "\t Everyone has full permissions.\n";
166. }
167. else
168. {
169.     foreach my $Permission ( sort( keys( %PermList ) ) )
170.     {
171.         foreach my $DaclType
172.             (
173.                 sort( keys( %{$PermList{$Permission}} ) ) )
174.             {
175.                 my $String = $PermList{$Permission}->{$DaclType};
176.                 printf( " % ". $MaxAcctLength . "s % -11s %s\n",
177.                         $Permission,
178.                         "($DaclType)",
179.                         join( ' ', @{$String} );
180.             }
181.     }

```

## Decoding Flags

When you have obtained the flags from a particular ACE, you can decode the flags into a more human readable format:

```
$Count = Win32::Perms::DecodeFlag( $Flag | $HashRef [, \@Flags] );
```

The first parameter (`$Flag` or `$HashRef`) is either the numeric flag value or a hash reference obtained with a call to the `Get()` method. It is preferred to pass in the hash reference because it contains information regarding how to interpret the flags. Passing in a flag value alone will only result in generic flags.

The optional second parameter (`\@Flags`) is a reference to an array. This array will be populated with a list of flag strings. Each string represents a particular flag that the passed-in flag value or hash reference has set.

Because of how often this function is used, the extension exports the function's name into the main namespace. This means you can call the function from a script in one of two ways:

```
Win32::Perms::DecodeFlag();
DecodeFlag();
```

The latter example is convenient because you don't have to type in the full namespace of the function's extension.

## Decoding Permission Types

When you have obtained the type value from a particular ACE, you can decode it into a more human readable format:

```
$Count = Win32::Perms::DecodeType( $Type | $HashRef [, \@Types] );
```

The first parameter (`$Type` or `$HashRef`) is either the numeric type value or a hash reference obtained with a call to the `Get()` method. It is preferred to pass in the hash reference because it contains information regarding how to interpret the type value. Passing in a type value alone will only result in generic type strings.

The optional second parameter (`\@Types`) is a reference to an array. This array will be populated with a list of type strings. Each string represents a particular ACE type that the passed-in type value or hash reference has set. Because of how often this function is used, the extension exports the function's name into the main namespace. This means you can call the function from a script in one of two ways:

```
Win32::Perms::DecodeType();
DecodeType();
```

The latter example is convenient because you don't have to type in the full namespace of the function's extension.

## Adding a User to a Security Object

There are different ways to add a user account to a `Win32::Perms` object. They are broken out by function:

```
$PermsObj->Allow( $Account [, $Mask [, $Flag] ] );
$PermsObj->Deny ( $Account [, $Mask [, $Flag] ] );
$PermsObj->Audit ( $Account [, $Mask [, $Flag] ] );
```

The first parameter (`$Account`) represents the user account you are adding. This is in any of the formats listed in [Table 11.1](#).

The optional second parameter (`$Mask`) indicates the permissions to be applied to the particular user. This can be any combination of values from [Table 11.2](#). If this parameter is not specified, a default permission value of `GENERIC_ALL` is used.

The optional third parameter (`$Flag`) indicates the ACE flags to be applied to the particular user. This can be any combination of values from [Table 11.3](#) and [Table 11.12](#). If this parameter is not specified, a default flag value of `CONTAINER_INHERIT_ACE` is used.

These methods will grant the specified user permissions (`Allow`), deny the specified user permissions (`Deny`), or Add an auditing ACE (`Audit`) for a specified user.

If the methods are successful, they return a `true` value (nonzero); otherwise, they fail and return a zero (`0`) value.

**Table 11.12. Additional Flags That Can be Passed into the Addxxxx() Methods**

Flag	Description
DIRECTORY DIR	Permission is for a directory.
KEY	Permission is for a Registry key.
CONTAINER	Permission is for container object.
FILE	Permission is for a file.
VALUE	Permission is for a Registry value.
NON_CONTAINER	Permission is for a noncontainer, such as a file or Registry value.
SUCCESS	Audit successful events. This only applies to System Audit types.
FAILURE	Audit failed events. This only applies to System Audit types.

[Example 11.10](#) shows how to use these different methods. Line 10 calls `Allow()` to add an ACE that will allow the user to access certain permissions. Line 6 checks to see if the object is a container or not; there will be more discussion on this in the "Checking For Containers" section later in this chapter. If it is a container (such as a directory), it calls line 10, which sets `READ` and `EXECUTE` permissions. The specified flag, though, is `CONTAINER_INHERITACE`. This causes the ACE to be inherited into all subcontainers. This means that ACE only applies to directories.

Line 22 calls the `Deny()` method. This will deny any access to change the object. The flags `(OBJECT_INHERITACE | INHERITONLYACE)` indicate that all objects inside the container will inherit the ACE (such as files and directories). It also means that a subdirectory that inherits the ACE will not apply it but only hold on to it so that other subobjects can inherit it. Line 26 calls the `Set()` method, which is described in the section "Setting Permissions on an Object" later in this chapter.

## Tip

*Often you will see two ACEs for the same user account on a container object. This is common because one ACE might apply to the container object, and the other ACE might apply to objects within the container.*

*Consider a directory. You might have certain permissions on the directory itself that don't really make sense for files. So you could add an ACE with only permissions for the directory. You would mark it with the `CONTAINER_INHERITACE` flag. This means that the ACE would only be inherited by a subdirectory if you moved one or created one inside the directory.*

*The other ACE would only be applied to files that are created in the directory. This ACE would have the `INHERITONLYACE` flag, which means it does not apply to the directory (the container object) but only applies to objects that inherit the ACE (such as files).*

## **Example 11.10 Adding permissions for a user**

```
01. use Win32::Perms;
02. my $Path = shift @ARGV || $0;
03. my $User = shift @ARGV || "Guest";
```

```

04. my $PermsObj = new Win32::Perms( $Path ) || die;
05. print $PermsObj->Path(), "\n";
06. if( $PermsObj->IsContainer() )
07. {
08.     # We are a container so make sure sub containers will
09.     # inherit this permission...
10.    if( $PermsObj->Allow( $User, READ | EXECUTE,
CONTAINER_INHERIT_ACE ) )
11.    {
12.        print "Successfully set the directory's permissions.\n";
13.    }
14. else
15.    {
16.        print "Failed to set the directory's permissions.\n";
17.    }
18. }
19. # Add the user's permission. Make sure that this will be
passed
20. # down to each new sub container and file, but it does not
apply to
21. # the current container object...
22. if( $PermsObj->Deny( $User, CHANGE, OBJECT_INHERIT_ACE
23.                         | INHERIT_ONLY_ACE ) )
24. {
25.    print "Successfully set the $User account to deny CHANGE
permissions.\n";
26.    $PermsObj->Set();
27. }
28. else
29. {
30.    print "Failed to set the $User account to allow CHANGE
permissions.\n";
31. }

```

## Adding a User to a Security Object, Part 2

In the preceding section, we looked at three different methods of adding a permission ACE to an object. They are fairly easy to use but not as flexible. Now it's time to introduce the `Add()` method. This is by far the most flexible way of adding an ACE:

```
$PermsObj->Add( $Account [, $Mask [, $Type [, $Flag ] ] ] )
$PermsObj->Add( \%Hash [, \%Hash2 [, ...] ] )
```

The first parameter (`$Account` or `\%Hash`) can either be a string representing a user account or a hash reference. If it is a hash reference, the format of the hash is expected to be from a call to the `Get()` method. Any number of these hash references can be passed into the method. However, if the parameter is a string, it must be in a format described in [Table 11.1](#).

The optional second parameter (`$Mask`) indicates the permissions to be applied to the particular user. This can be any combination of values from [Table 11.2](#). If this parameter is not specified, a default permission value of `GENERIC_ALL` is used.

The optional third parameter (`$Type`) indicates the ACE type. This can be any one value from [Table 11.4](#). Alternatively, you can specify a value from [Table 11.13](#) instead. If this parameter is not specified, a default permission value of `ACCESS_ALLOWED_ACE_TYPE` is used.

The optional fourth parameter (`$Flag`) indicates the ACE flags to be applied to the particular user. This can be any combination of values from [Table 11.3](#) and [Table 11.12](#). If this parameter is not specified, a default flag value of `CONTAINER_INHERITACE` is used.

The method returns the number of ACEs that were successfully added.

**Table 11.13. Types That Can Be Passed into the `Add()` Method**

Type	Description
<code>ALLOW</code>	Permission grants allow access.
<code>GRANT</code>	
<code>DENY</code>	Permission mask denies access.
<code>AUDIT</code>	Permission mask describes what to audit. The ACE flags will indicate whether to audit successes or failures.

In [Example 11.11](#), we demonstrate how using the `Add()` method is similar to using the previous methods. Notice that this example is almost identical to [Example 11.10](#). The difference is that lines 10 and 23 use `Add()` instead of `Allow()` and `Deny()`. As with the other example, line 28 calls the `Set()` method, which is described in the section "[Setting Permissions on an Object](#)" later in this chapter.

### **Example 11.11 Adding permissions using the `Add()` method**

```
01. use Win32::Perms;
02. my $Path = shift @ARGV || $0;
03. my $User = shift @ARGV || "Guest"; mbered">
04. my $PermsObj = new Win32::Perms( $Path ) || die;
05. print $PermsObj->Path(), "\n";
06. if( $PermsObj->IsContainer() )
07. {
08.     # We are a container so make sure sub containers will
09.     # inherit this permission...
10.    if( $PermsObj->Add( $User, READ | EXECUTE,
ACCESS_ALLOWED_ACE_TYPE,
11.
CONTAINER_INHERITACE ) )
12.    {
13.        print "Successfully set the directory's permissions.\n";
14.    }
15. else
16. {
```

```

17.     print "Failed to set the directory's permissions.\n";
18. }
19. }
20. # Add the user's permission. Make sure that this will be
passed
21. # down to each new sub container and file, but it does not
apply to
22. # the current container object...
23. if( $PermsObj->Add( $User, CHANGE, ACCESS_DENIED_ACE_TYPE,
24.                      OBJECT_INHERIT_ACE
25.                      | INHERIT_ONLY_ACE ) )
26. {
27.     print "Successfully set the $User account to deny CHANGE
permissions.\n";
28.     $PermsObj->Set();
29. }
30. else
31. {
32.     print "Failed to set the $User account to allow CHANGE
permissions.\n";
33. }

```

Because the `Add()` method can also be passed in hash references, this makes it the ideal mechanism to copy permissions from one object to another. The array of hashes generated by the `Get()` method can be passed directly into the `Add()` method without any modifications. This is what [Example 11.12](#) does. Notice that line 7 calls `Get()` to populate the `@List` hash. When done, line 8 removes all ACEs in the destination `Win32::Perms` object (`$DestPerms`); there is more discussion on this in the next section, "Removing Permissions."

Finally, a call to `Add()` is performed, passing in the array. Because the array itself is passed in (not an array reference), the array expands so that each element in the array becomes a separate parameter passed into the method. Because each of the array entries represents a hash reference with all of the required keys and values, the permissions are effectively copied from `$OrigPerms` to `$DestPerms`.

Line 10 commits the new permissions settings to the object by calling the `Set()` method. There will be more discussion on this later in the section "Setting Permissions on an Object."

### ***Example 11.12 Copying permissions from one object to another***

```

01. use Win32::Perms;
02. my $OrigPath = shift @ARGV || die;
03. my $DestPath = shift @ARGV || die;
04. my $OrigPerms = new Win32::Perms( $OrigPath ) || die;
05. my $DestPerms = new Win32::Perms( $DestPath ) || die;
06. my @List;
07. $OrigPerms->Get( \@List );
08. $DestPerms->Remove( -1 );
09. $DestPerms->Add( @List );
10. if( $DestPerms->Set() )
11. {

```

```

12.     print "Successfully copied permission from:\n";
13.     print "  $OrigPath\n";
14.     print "to:\n";
15.     print "  $DestPath\n";
16.   }
17. else
18. {
19.   print "Failed to copy permissions.\n";
20. }
```

## Removing Permissions

When you have a `Win32::Perms` object, you might want to remove ACEs. After all, you don't always want to just add ACEs to the security of an object. The `Remove()` and `RemoveAudit()` methods do this:

```

$Total = $PermsObj->Remove( -1 );
$Total = $PermsObj->Remove( $Account | $Index [, $Account2 |
$Index2 [, ... ] ] );
$Total = $PermsObj->RemoveAudit( -1 );
$Total = $PermsObj->RemoveAudit( $Account | $Index [, $Account2 |
$Index2 [, ... ] ]
] );
```

Both of the methods (`Remove()` and `RemoveAudit()`) behave identical to each other. The only difference between them is that the former remove ACEs from the DACL (hence removing permissions). The latter removes ACEs from the SACL (hence removing auditing information).

If the first and only parameter to either of these methods is a `-1` value, all of the ACEs are removed. This ends up with an empty DACL or SACL. This will end up denying everyone (except the owner, of course) access to the object (if `Remove()` was called) or will turn off all auditing (if `RemoveAudit()` was called).

Otherwise, the different parameters can consist of either a string representing a user account or a numeric value indicating the index number of which ACE to remove. If the parameter is a string, it must follow one of the formats listed in [Table 11.1](#). In this case, *every* ACE that matches the passed-in user account will be removed.

If a parameter is a numeric value (that is not `-1`), it corresponds to an array element that was obtained by calling `Get()`.

[Example 11.13](#) shows how you can selectively remove a particular user from the list of permissions. Lines 6 and 7 remove all permissions and auditing information for the specified user.

### ***Example 11.13 Removing permissions for a user account***

```

01. use Win32::Perms;
02. my $Path = shift @ARGV || $0;
03. my $User = shift @ARGV || Win32::DomainName() . "\\\Guest";
04. my $PermsObj = new Win32::Perms( $Path ) || die;
```

```

05. print $PermsObj->Path(), "\n";
06. $iTotal = $PermsObj->Remove( $User );
07. $iTotal += $PermsObj->RemoveAudit( $User );
08. if( $PermsObj->Set() )
09. {
10.   print "Successfully removed $iTotal ACEs for $User.\n";
11. }
12. else
13. {
14.   print "Failed to remove any ACEs for $User.\n";
15. }

```

To better exemplify the capabilities of the remove methods, look at [Example 11.14](#). In this case, line 7 returns the total number of permissions (ACEs) in the `Win32::Perms` object. It also populates the `@List` array with a hash reference for each ACE. Lines 11 through 20 walk through each ACE in the list, looking for an inherited permission (line 16). If it finds one, the permission is then removed (line 18). Notice that we remove the permission by specifying the index entry of the ACE in the `@List` array that was set by the `Get()` method.

The most important aspect of [Example 11.14](#) is that lines 11 through 20 count *down* when processing each ACE in the list. It starts with the last

ACE entry in the list and work its way to the first entry. This is very important because if you were to remove ACE entry 1, entry 2 becomes entry 1. Therefore, if you attempt to remove the next entry (at index number 2), you would now be removing what was originally entry 3. Effectively, you just skipped entry 2.

### ***Example 11.14 Removing inherited permissions***

```

01. use Win32::Perms;
02. my $Path = shift @ARGV || $0;
03. my $User = shift @ARGV || Win32::DomainName() . "\\" Guest";
04. my $PermsObj = new Win32::Perms( $Path ) || die;
05. my @List;
06. my $iTotal = 0;
07. my $iIndex = $PermsObj->Get( \@List );
08. print $PermsObj->Path(), "\n";
09. # Make sure we could *down* so that we don not skip
10. # any entries when removing them.
11. while( $iIndex- )
12. {
13.   my $Ace = $List[$iIndex];
14.   # Remove any permissions that have been inherited.
15.   # This only applies to Windows 2000/XP platforms.
16.   if( $Ace->{Flag} && INHERITED_ACE )
17.   {
18.     $iTotal += $PermsObj->Remove( $iIndex );
19.   }
20. }
21. if( $iTotal )
22. {

```

```

23. if( $PermsObj->Set() )
24. {
25.     print "Successfully removed $iTotal inherited
permissions.\n";
26. }
27. else
28. {
29.     print "Failed to remove any ACEs for $User.\n";
30. }
31. }
32. else
33. {
34.     print "There were no inherited permissions to remove.\n";
35. }

```

## Setting Permissions on an Object

When you have modified a permissions object as much as you need to, you have to commit the changes for it to have any effect on the secured object.

This is what the various `Setxxxxx()` methods do. Until you call one of these methods, none of your changes will actually be stored. Therefore, the changes will have no effect until you call one of the `Setxxxxx()` methods:

```

$PermsObj->Set( [ $Path [, $Recurse [, $Container ] ] ] );
$PermsObj->SetDacl( [ $Path [, $Recurse [, $Container ] ] ] );
$PermsObj->SetSacl( [ $Path [, $Recurse [, $Container ] ] ] );
$PermsObj->SetOwner( [ $Path [, $Recurse [, $Container ] ] ] );
$PermsObj->SetGroup( [ $Path [, $Recurse [, $Container ] ] ] );

```

The optional first parameter (`$Path`) is the path to a securable object. This enables you to set permissions on any valid path. The list of valid path formats is in [Table 11.10](#). If this parameter is not specified, the permissions are set on the object returned by a call to the `Path()` method.

Generally, this is the path specified when you created the `Win32::Perms` object. If no path was specified when creating the object and no path is passed into this method, the method will fail.

The optional second parameter (`$Recurse`) is a Boolean value that tells the `Win32::Perms` object to commit the permissions to every object that matches the path criteria. It will then recurse into subcontainers and set matching objects as well. For example, if the path is `C:\TEMP\*.*` and `$Recurse` is nonzero, then permissions will be applied to every file or directory of `C:\TEMP\`, depending on the container state of the `Win32::Perms` object. Refer to the section "[Understanding Setting Permissions Recursively](#)" for more details.

The optional third parameter (`$Container`) specifies whether or not to interpret the `Win32::Perms` object as a container. When setting permissions by passing in a path, only objects that match the same container state of the `Win32::Perms` object are set. This means that if a `Win32::Perms` object represents a directory, only directories will be set. This flag lets you override the current setting so that a directory `Win32::Perms` object can set permissions on a file. See the section "[Understanding Setting Permissions Recursively](#)" for more details.

The `Set()` method will set all permissions, security auditing information, the owner, and the primary group on an object. The `SetDacl()` method will only set the DACL (or the security permissions) on an object. The `SetSacl()` method will only set the SACL (or the security auditing information) on an object. The `SetOwner()` method will only set the owner on an object. The `SetGroup()` method will only set the primary group on an object.

If no path is passed into these methods, the settings will be set on the default path, which is queryable by calling `$PermsObj->Path()`. This is generally the path that was used to create the `Win32::Perms` object.

The method will return the number of objects that it successfully set.

Example 11.15 uses both the `Set()` method (line 15) and the `SetRecurse()` method (line 9, described in the "Recursively Setting Permissions" section). Both methods are functionally the same.

### **Example 11.15 Storing permissions on multiple files**

```
01. use Win32::Perms;
02. my $Path = shift @ARGV || ".";
03. my $User = shift @ARGV || Win32::DomainName() . "\Guest";
04. my $PermsObj = new Win32::Perms( $Path ) || die;
05. print $PermsObj->Path(), "\n";
06. $PermsObj->Remove( $User );
07.
08. # Set permissions on only files, not directories.
09. $iTotal = $PermsObj->SetRecurse( "$Path\*\.*", 0 );
10. print "Set permissions on $iTotal files.\n";
11.
12. # Set permissions on only directories, not files.
13. # This time use Set() so we have to pass in TRUE for
14. # both recursion (parameter 2) and container (parameter 3).
15. $iTotal = $PermsObj->Set( "$Path\*\.*", 1, 1 );
16. print "Set permissions on $iTotal directories.\n";
```

Example 11.16 illustrates the use of a different set method. Here we create a new *empty* `Win32::Perms` object. Notice that there is no path passed into the `new()` function on line 4. Line 7 then sets the owner on the object. If it is successfully set, we check for any wildcards in the path. If we find some, we assume that we want to recurse into subcontainers. Line 11 sets only the owner on the noncontainer objects specified by the path. Line 13 sets only the owner on container objects.

### **Example 11.16 Recursively changing file ownership**

```
01. use Win32::Perms;
02. my $Path = shift @ARGV || "*.*";
03. my $User = shift @ARGV || die;
04. my $PermsObj = new Win32::Perms() || die;
05. print $PermsObj->Path(), "\n";
06. print "Setting owner to: $User\n";
07. if( "" ne $$PermsObj->Owner( $User ) )
08. {
09.     my $Recurse = ( $Path =~ /[^\*\?]/ );
```

```

10.    # First set files...
11.    my $iFiles = $PermsObj->SetOwner( $Path, $Recurse, 0 );
12.    # Second set directories...
13.    my $iDirs = $PermsObj->SetOwner( $Path, $Recurse, 1 );
14.    print "Set owner on $iFiles files and $iDirs directories.\n";
15. }
16. else
17. {
18.     print "Unable to set the owner.\n";
19. }

```

## Recursively Setting Permissions

Much like setting permissions with the `Set()` method, you can also set permissions recursively. This results in all objects matching specified criteria:

```
$PermsObj->SetRecurse( $Path [, $Container ] );
```

The first parameter (`$Path`) is the path to a securable object. This enables you to set permissions on any valid path. The list of valid path formats is in [Table 11.10](#). If this parameter is not specified, the permissions are set on the object returned by a call to the `Path()` method. Generally, this is the path specified when you created the `Win32::Perms` object. If no path was specified when creating the object and no path is passed into this method, the method will fail. This path can include wildcards.

The optional third parameter (`$Container`) specifies whether or not to interpret the `Win32::Perms` object as a container. When setting permissions by passing in a path, only objects that match the same container state of the `Win32::Perms` object are set. This means that if a `Win32::Perms` object represents a directory, only directories will be set. This flag lets you override the current setting so that a directory `Win32::Perms` object can set permissions on a file.

Calling this method is no different from calling the `Set()` method passing in a path, a `true` value for the `$Recurse` parameter, and (optionally) a container flag.

The method will return the number of objects that it successfully set.

### ***Understanding Setting Permissions Recursively***

It is very important to understand what happens when you are setting permissions while specifying wildcards and/or recursing into subcontainers.

### ***Matching Container Types***

By far, the most confusing thing about setting permissions using wildcards and/or recursion is that only objects that match the `Win32::Perms` object's container type are set.

When you create a `Win32::Perms` object by providing a path in the `new()` function, the extension checks whether or not the object is a container. Whatever the state is, the object remembers it. So if you create a `Win32::Perms` object from a path of `C:\TEMP\` or `HKEY_LOCAL_MACHINE\Software`, it will remember that it represents a container.

Permissions will *only be set on objects with the same container state*. Therefore, the `Win32::Perms` object we created for a container cannot be set on the file `C:\TEMP\Test.txt`. This is done so you can control what objects are affected when you set permissions.

All the `Setxxxxx()` methods enable you to pass in a container flag. This enables a script to override the `Win32::Perms` object's container setting temporarily. Using this flag enables you to set permissions on files even though the `Win32::Perms` object represents a directory.

## **Wildcards**

When wildcards are specified in a path (such as `C:\TEMP\*.*`), `Win32::Perms` will consider the last component of the path (`*.*`) to be the target for which to search. It will start its search in the first part of the path (`C:\TEMP\`). Therefore, specifying a path of `C:\TEMP\*.*` will set all objects in the `C:\TEMP` directory, but specifying `C:\TEMP` will only set the `TEMP` object inside the `C:\` directory.

The two wildcards that are recognized are the `*` and `?` characters. These are standard DOS/Windows wildcards. However, wildcards are only respected by file system objects (files and directories). This means that other objects (Registry keys, network shares, printers, and so on) do not support wildcards.

## **Recursion**

When recursion is specified, every subcontainer will be traversed, and all objects within it will be examined for a match. If the subobjects match the wildcard specification or the path, the object will be set.

Now, this is not as obvious as it might sound. For example, if a path of `C:\TEMP` is specified, it assumes that `TEMP` is the target (as mentioned in the preceding section, "Wildcards"). However, if recursion is enabled, every directory in `C:\` will be searched for an object called `TEMP`. Then each of the subdirectories will also be searched.

Therefore, it is possible that with a path of `C:\TEMP` and recursion set, permissions will be set on the following:

```
C:\TEMP  
C:\Winnt\System32\Temp  
C:\Program Files\Temp
```

This is because `C:\` and all of its subdirectories are searched for its target of `TEMP`. If this still seems odd to you, consider if you specified a path of `C:\*.*`. In this case, you would be saying to examine `C:\` for any file.

If the specified path is a relative path (such as `..\test.txt`), the path is considered relative to the current working directory. You can query what current working directory is using `Win32::GetCwd()` (refer to [Chapter 9](#) for more details on querying and changing the current working directory).

## **Limitations of Recursing with**

# Wildcards

*Paths that specify a Registry key cannot contain wildcards but can use recursion.  
Recursion does not apply to network or printer shares.*

## Checking For Containers

If you have a `Win32::Perms` object, you might want to discover whether or not it is a container (such as a directory or Registry key). Such information is important when committing the following:

```
$PermsObj->IsContainer( [$Container] );
```

The optional parameter (`$Container`) is a Boolean value of either `TRUE` (1) or `FALSE` (0). This parameter will set the container state of the `Win32::Perms` object.

The method returns a `TRUE` (1) or `FALSE` (0) value, indicating whether the `Win32::Perms` object represents a container.

[Example 11.17](#) shows how you can check whether a `Win32::Perms` object represents a container.

### **Example 11.17 Checking for containers**

```
01. use Win32::Perms;
02. my $Path = shift @ARGV || ".";
03. $Path = Win32::GetFullPathName( $Path );
04. my $PermsObj = new Win32::Perms( $Path ) || die;
05. print $PermsObj->Path(), "\n";
06. print "This is a ", $PermsObj->GetType(), " which is ";
07. print (($PermsObj->IsContainer())? ":""not " ) . "a
container.\n";
```

## Importing Security Descriptors

When you have a `Win32::Perms` object, you can import permissions from various different objects. The `Import()` method does this:

```
$PermsObj->Import( $Path | $Object | $SD );
```

The only parameter can either be a path to a securable object or a recognized Perl object. If a path is specified, it must be in one of the formats found in [Table 11.10](#). Additionally, blessed Perl objects can be passed into this function. For example, a `Win32::Registry` object can be passed in. This would cause the `Win32::Perms` object to reflect the permissions of the Registry key. [Example 11.3](#) demonstrates this.

You can also pass in a binary security descriptor. An example of such a descriptor can be found in the Registry.

The method returns the number of ACEs that were imported from the security descriptor.

An interesting way to show how security descriptors can be imported is in [Example 11.18](#). In this case, the script extracts a binary security descriptor from the Registry. In particular, the script removes guest access from DCOM's default access SD. By modifying this SD, you can allow or deny a user DCOM access to a machine. This is normally configured using the `DCOMCNFG.EXE` program.

In line 7, the script opens the required Registry key and queries for the `DefaultAccessPermission` value (line 12). This is a binary security descriptor. Line 17 imports the SD into a `Win32::Perms` object. Then lines 19 through 22 remove any unwanted user access. Line 24 exports a relative binary security descriptor and line 26 stores it back into the Registry.

Line 33 cleans out the `Win32::Perms` object so that it is empty, and then the script again queries the Registry for the updated SD. This time, instead of modifying the SD, it is dissected (line 40), and the different user accounts are displayed (lines 43 through 47).

### ***Example 11.18 Modifying DCOM's default access permission list***

```
01. use Win32::Registry;
02. use Win32::Perms;
03. my $PermsObj = new Win32::Perms() || die "Cannot create
Permissions object";
04. my $Key;
05. my $Domain = Win32::DomainName();
06. $Domain .= "\\\" unless( "" eq $$Domain );
07. if( $HKEY_LOCAL_MACHINE->Open( "Software\\Microsoft\\OLE",
$key ) )
08. {
09.     my( $Flag, $Type, $Value, $Sd );
10.    # Make sure to undef $Value otherwise QueryValueEx() gets
upset...
11.    undef $Value;
12.    if( ( $Key->QueryValueEx( "DefaultAccessPermission", $Type,
$Sd ) )
13.        && ( REG_BINARY == $Type ) )
14.    {
15.        my @List;
16.        # Import the SD's ACEs...
17.        $PermsObj->Import( $Sd );
18.        # Remove 'Everyone' and Guests....
19.        $PermsObj->Remove( "Guest" );
20.        $PermsObj->Remove( $Domain . "Guest" );
21.        $PermsObj->Remove( $Domain . "Domain Guests" );
22.        $PermsObj->Remove( "Everyone" );
23.        # Export the binary SD...
24.        $Sd = $PermsObj->GetSD( SD_RELATIVE );
25.        # Store the binary SD back into the Registry...
26.        if( ! $Key->SetValueEx( "DefaultAccessPermission", 0,
REG_BINARY, $Sd ) )
27.    {
28.        print "Failed to set the new security descriptor.";
```

```

29.         print "Error: ",
Win32::FormatMessage( Win32::GetLastError() ), "\n";
30.         exit;
31.     }
32.     # Clean out the permission list...
33.     $PermsObj->Remove( -1 );
34.     # Re-query the SD...
35.     if( ( $Key->QueryValueEx( "DefaultAccessPermission", $Type,
$Sd ) )
36.         && ( REG_BINARY == $Type ) )
37.     {
38.         # Import the SD again...
39.         $PermsObj->Import( $Sd );
40.         if( $PermsObj->Get( \@List ) )
41.         {
42.             print "The following are allowed to access this
machine using DCOM:\n";
43.             foreach my $Ace ( @List )
44.             {
45.                 my $Domain = "$Ace->{Domain}\" if( "" ne $Ace-
>{Domain} );
46.                 print "\t", $Domain, $Ace->{Account}, "\n";
47.             }
48.         }
49.         else
50.         {
51.             print "No one is allowed to access this machine using
DCOM.\n";
52.         }
53.     }
54. }
55. else
56. {
57.     print "Unable to access the DCOM security descriptor.\n";
58.     print "Run DCOMCNFG.EXE to make sure that there are ";
59.     print "some default security settings.\n";
60. }
61. $Key->Close();
62. }
63. $PermsObj->Close();

```

## Exporting a Security Descriptor

There are times when you might want to export a security descriptor from a `Win32::Perms` object. [Example 11.18](#) is a perfect example of this. Likewise, when you are creating a new network share, you can pass in a security descriptor specifying the actual permissions for the share (refer to [Chapter 2](#), "Network Administration"). Anywhere function takes in a security descriptor, you can use a `Win32::Perms` object to export one.

The actual technique to exporting an SD is fairly simple. You simply have to call the `GetSD()` method:

```
$Sd = $PermsObj->GetSD( [ SD_RELATIVE | SD_ABSOLUTE ] );
```

The only parameter the method accepts is either the `SD_RELATIVE` or `SD_ABSOLUTE` constant. The constant indicates what type of security descriptor to export. If no constant is specified, `SD_ABSOLUTE` is assumed.

If a relative SD is requested, the method returns a binary block of memory that represents an SD. Perl handles this SD as if it were a string.

If an absolute SD is requested, a numeric value is returned from the method. This value indicates where in memory the absolute SD exists. This is basically a pointer to the SD in memory.

## Warning

*Using absolute SDs can be dangerous. If you export an absolute SD and then modify the Win32::Perms object in any way, the absolute SD might no longer be valid. However, if you continue to use the exported absolute SD value after the Win32::Perms object has been modified, the script might crash. This is unavoidable, so care must be taken.*

[Example 11.18](#) is a good example of using the `GetSD()` method. Line 24 calls this method to procure a self-relative security descriptor.

## Exporting ACLs

Just as you can export a security descriptor, you can also export an Access Control List (ACL). This is done with these different methods:

```
$Acl = $PermsObj->GetDacl( ACL_RELATIVE | ACL_ABSOLUTE );
$Acl = $PermsObj->GetSacl( ACL_RELATIVE | ACL_ABSOLUTE );
```

Both of these methods must pass in a parameter indicating the type of ACL to export.

If a relative ACL is requested, the method returns a binary block of memory that represents an ACL. Perl handles this ACL as if it were a string.

If an absolute ACL is requested, a numeric value is returned from the method. This value indicates where in memory the absolute ACL exists. This is basically a pointer to the ACL in memory.

## Warning

*Using absolute ACLs can be dangerous. If you export an absolute ACL and then modify the Win32::Perms object in any way, the absolute ACL might no longer be valid. However, if you continue to use the exported absolute ACL value after the Win32::Perms object has been modified, the script might crash. This is unavoidable, so care must be taken.*

The methods will return either a Discretionary Access Control List or a System Access Control List in either absolute or relative format.

## Checking for Valid SDs and ACLs

Because the `Win32::Perms` extension can export and import both security descriptors and ACLs, it is worth mentioning that it can also check their validity as well. By using the `CheckSD()` and `CheckACL()` functions, a script can determine whether a piece of memory is properly formatted as an SD or ACL:

```
Win32::Perms::CheckSD( $Sd );
Win32::Perms::CheckACL( $Acl );
```

The only parameter these methods accept is a relative or absolute security descriptor or an ACL, depending on which method is being called.

If the passed-in object is indeed verified to be valid, the methods return a `TRUE (1)` value. Otherwise, the methods return `FALSE (0)`.

## Using `Win32::FileSecurity`

When I started to write the first edition of this book, the only real Win32 security extension available was `Win32::FileSecurity`. It focused solely on file-based permissions. Since then, `Win32::Perms` has matured into a very useful extension that covers much more than just NTFS file permissions. And `Win32::FileSecurity` seems to be used less and less. However, because many users still rely on the extension (after all, it is quite useful), this section is devoted to the use of the `Win32::FileSecurity` extension.

Considering how complicated file security can be, the `Win32::FileSecurity` extension is remarkably easy to learn and use. It provides a basic interface to retrieve, set, and decode permissions.

To use the extension, you must, as with all extensions, `use` it:

```
use Win32::FileSecurity;
```

## Using Constants

Early on in the development of the `Win32::FileSecurity` extension, constants were resolved in an unconventional manner. However, this was way back in the days of Win32 Perl v5.003. Things have changed since then, but originally, the `Win32::FileSecurity` extension did not export the constants from the extension's module into the main namespace. This meant you could not just use a constant by specifying its name. For example

```
$Mask = STANDARD_RIGHTS_READ | STANDARD_RIGHTS_WRITE;
```

would not work. You had to specify the constant by its full namespace; to make matters worse, you had to specify the constant as a function, as in the following:

```
$Mask = Win32::FileSecurity::STANDARD_RIGHTS_READ() |  
Win32::FileSecurity::STANDARD_RIGHTS_WRITE();
```

The reason for this was that the constants were supported by the extension, but the module did not export the list of constant names. Because the names are not exported into the main namespace, Perl did not know what the constant was or how to resolve it. When you attempted to call a function in the `Win32::FileSecurity` extension that had the name of a constant, it forced Perl to resolve the function's name. By doing this, the constant's value was resolved instead. Therefore, don't be surprised if you see older code that specifies the full namespace of a constant or if a constant is referenced as a function.

After a constant has been resolved, the value is cached, and you can later refer to the constant by its full namespace name and not a function, as in [Example 11.19](#).

### **Example 11.19 Accessing a `Win32::FileSecurity` constant**

```
01. use Win32::FileSecurity;  
02. my $Mask = Win32::FileSecurity::READ();  
03. if( $Mask & READ )  
04. {  
05.   print "The mask contains the read permission\n";  
06. }
```

An alternative is to use the `Win32::FileSecurity::constant()` function:

```
$Value = Win32::FileSecurity::constant( $Permission, $Value );
```

The first parameter (`$Permission`) is a string representation of a constant such as "`ADD`". Refer to Table 4.10 for a list of constants.

The second parameter (`$Value`) can be anything, but it must be passed in.

If the `Win32::FileSecurity::constant()` function is successful, the value of the constant is returned; otherwise, a `0` is returned. This can be quite a problem because it is impossible to determine whether a constant's value is `0` or whether there is no such constant.

## **The Security Hash**

The `Win32::FileSecurity` extension utilizes a security hash. This hash consists of keys named after a user or group account name. The value associated with a key is the permission mask (see the following section, "The `Win32::FileSecurity` Mask"). An example of such a security hash dump is as follows:

```
%Perms = (  
  'BUILTIN\\Administrators' => 2032127  
  'rothd' => 1245631
```

```
'CREATOR OWNER' => 2032127
'Everyone' => 1245631
'NT AUTHORITY\\SYSTEM' => 2032127
)
```

Notice that the keys are legitimate user or group names, and the values are numeric masks that can be decoded using the `EnumerateRights()` function (see the later section "Decoding a `Win32::FileSecurity` Permission Mask").

## The `Win32::FileSecurity` Mask

When dealing with permissions, you will come across the concept of a mask. A permission mask is a bitmask of permission values all logically OR'ed together.

When you set permissions, you need to assign a mask to a user or group account. You can create a hash either by OR'ing the constant values together (refer to the earlier section "Using Constants" for details on using constants in this extension) or by using the `Win32::FileSecurity::MakeMask()` function:

```
$Mask = Win32::FileSecurity::MakeMask( $Permission [, $Permission2
[, ... ] ] );
```

Any number of permission strings can be passed into the `MakeMask()` function. The permissions passed in are the string versions of those listed in [Table 11.10](#).

### Note

*When using the `MakeMask()` function, it is very important to pass in the names of the constants as strings. To create a mask of `READ` and `DELETE` access, you would use the following:*

```
$Mask = Win32::FileSecurity::MakeMask( "READ", "DELETE" );
```

*In older versions of the extension, you could specify only the constant name, but this will not work in the newer versions.*

*Newer versions export the constant names, so passing in just the name of the constant will result in the numeric values being passed in. This indeed would not work well.*

## Retrieving File Permissions

When you need to retrieve the permissions from a file or directory, you can use the `Win32::FileSecurity::Get()` function:

```
Win32::FileSecurity::Get( $Path, \%Perms );
```

The first parameter (`$Path`) is the path to the object for which you want security information. This can be a relative path, a full path, or a UNC.

The second parameter (`\%Perms`) is a reference to a hash. If the function is successful, this hash will be populated with keys that represent the user or group account. The values of the keys will be the permission mask for that account.

If the `Get()` function is successful, it will return `true`; otherwise, it returns `false`.

## Decoding a `Win32::FileSecurity` Permission Mask

After you have retrieved the permission hash using the `Get()` function, you can decode the mask by using the `Win32::FileSecurity::EnumerateRights()` function:

```
Win32::FileSecurity::EnumerateRights( $Mask, \@Permissions );
```

The first parameter (`$Mask`) is the bitmask that was either created with the `MakeMask()` function or retrieved by the `Get()` function.

The second parameter (`\@Permissions`) is a reference to an array. Strings representing the permissions that make up the mask will populate this array. These strings can be used to feed into the `MakeMask()` function. If the function is successful, it will return a `1`; otherwise, it returns a `0`. [Example 11.20](#) demonstrates the `EnumerateRights()` function.

### ***Example 11.20 Displaying account permissions on a file***

```
01. use Win32::FileSecurity;
02. my $Path = shift @ARGV;
03. my %Perms;
04. if( Win32::FileSecurity::Get( $Path, \%Perms ) )
05. {
06.     print "Permissions for $Path:\n";
07.     foreach my $Account (sort( keys( %Perms ) ) )
08.     {
09.         my @List;
10.         print "\t$Account:\n";
11.         if( Win32::FileSecurity::EnumerateRights( $Perms{$Account} ,
12. \@List ) )
13.         {
14.             map { print "\t\t$_\n"; } ( @List );
15.         }
16.         else
17.         {
18.             print "\t\tNone\n";
19.         }
20.     }
21. else
22. {
23.     print "Error accessing permissions for '$Path':\n";
24.     print Win32::FormatMessage( Win32::GetLastError() );
```

```
25. }
```

## Applying Masks to Directories or Files

When you have a mask that you want to apply to a directory or file, you use the `Win32::FileSecurity::Set()` function:

```
Win32::FileSecurity::Set( $Path, \%Permissions );
```

The first parameter (`$Path`) is a path to a file or directory. This parameter can be a UNC.

The second parameter (`\%Permissions`) is a reference to a security hash.

If the `Set()` function is successful, the permissions are placed on the file or directory and `true` is returned; otherwise, `false` is returned.

### **Case Study: Directory Security**

I have occasionally opened my server up to guests for various reasons. Because of this, I have had to secure particular directories from alterations by the anonymous users. The script listed in [Example 11.21](#) removes all write permissions for the Guest account and Guests group from the files in the `C:\TEMP` directory.

#### ***Example 11.21 Removing write permissions for a user on all files in a directory***

```
01. use Win32::FileSecurity;
02. my $Dir = "c:/temp";
03. my $User = "Guest";
04. my %Total;
05. print "Removing the WRITE permissions for $User in
directory:\n";
06. print "$Dir\n";
07. while( my $Path = glob( "$Dir/*.*" ) )
08. {
09.     my %Perms;
10.    my $Changes;
11.    next if (-d $Path);
12.    $Total{files}++;
13.    my( $File ) = ( $Path =~ /([^\\\/]*)$/ );
14.    print "\t$Total{files}) $File...";
15.    # Get the permissions of the file
16.    if( Win32::FileSecurity::Get( $Path, \%Perms ) )
17.    {
18.        foreach my $Account ( keys( \%Perms ) )
19.        {
20.            # If we have found the user then modify the permissions
21.            $Changes = ModifyPerms( \%Perms{$Account} )
22.            if( $Account =~ /$User$/i );
23.        }
24.        if( $Changes )
```

```

25.      {
26.          # If changes have been made then apply the new
permissions
27.          if( Win32::FileSecurity::Set( $Path, \%Perms ) )
28.          {
29.              print "\t\tSuccessful.\n";
30.              $Total{success}++;
31.          }
32.          else
33.          {
34.              print "\t\tUnable to set the new permissions for
$Path.\n";
35.              Error();
36.          }
37.      }
38.      else
39.      {
40.          print "\t\tNo changes were made.\n";
41.      }
42.  }
43.  else
44.  {
45.      print "Unable to get the permissions for $Path.\n";
46.      Error();
47.  }
48. }
49. print "\nTotal files: $Total{files}\n";
50. print "Total changes: $Total{success}\n";
51.
52. sub ModifyPerms
53. {
54.     my( $Mask ) = @_;
55.     my( @List, $Remove );
56.     # Decode the permission mask.
57.     if( Win32::FileSecurity::EnumerateRights( $$Mask, \@List ) )
58.     {
59.         for( my $Temp = $#List; $Temp >= 0; $Temp-- )
60.         {
61.             # If the following permissions exist then pop them
62.             # out of the permission array.
63.             if( $List[ $$Temp ] eq "WRITE" ||
64.                 $List[ $$Temp ] eq "STANDARD_RIGHTS_WRITE" ||
65.                 $List[ $$Temp ] eq "GENERIC_WRITE" ||
66.                 $List[ $$Temp ] eq "ADD" ||
67.                 $List[ $$Temp ] eq "CHANGE" )
68.             {
69.                 splice( @List, $Temp, 1 );
70.                 $Remove++;
71.             }
72.         }
73.         # Recreate the permission mask.
74.         $$Mask = Win32::FileSecurity::MakeMask( @List );

```

```

75.    }
76.  else
77.  {
78.    print "\t\tNo permissions for $Account were found.\n";
79.  }
80.  return $Remove;
81. }
82.
83. sub Error
84. {
85.   my( $Path ) = @_;
86.   print Win32::FormatMessage( Win32::GetLastError() );
87. }

```

Line 16 retrieves a hash containing accounts and permission masks for a file using the `Get()` function. The hash is scrutinized for a specified account. (In this example, the `GUEST` account is targeted.) For each of them, the permission bitmask is passed into a `ModifyPerms()` subroutine. This routine walks through the hash and picks apart the permission bitmask using the `EnumerateRights()` function, removing any permission that would allow writing. If any changes are made, they are applied to the file using the `Set()` function in line 27.

## LSA Functions

There is a little known part of all Windows NT/2000/XP machines that is known as the local security authority (LSA). This protected subsystem is responsible for maintaining the security of a computer. All aspects of security eventually go through some low-level LSA functions.

The set of LSA functions provides access into the user database, also known as the security accounts manager (SAM). Through these functions, a script can directly access user privileges, domain trusts, and other low-level security details. The `Win32::Lanman` extension provides access to these functions.

## User Privileges

Every Windows NT/2000/XP machine has a set of *user privileges* (sometimes referred to as *user rights*) that it can assign to users and groups. These privileges define what the account or group is permitted to do. For example, if an account is granted the Logon As A Service privilege, services are allowed to run using that specific account. If a group is granted the Remote Shutdown privilege, all members of the group are allowed to shut down the machine from a remote computer (using `Win32::InitiateSystemShutdown()`).

Privileges are not transitory; they relate only to a local machine. Therefore, even though a user is granted a permission on machine A, she might not be granted the same privilege on machine B. Likewise, a local machine's privileges are not affected by domain settings. Thus, if a user is granted a privilege on a domain controller, it does not mean she has the privilege on machines in the domain. Each member of the domain must grant her the privilege.

### Tip

*Because privileges are local to each machine, they can become difficult to manage on a domain scale. For example, let's say you have a service installed on every computer that runs under a domain account. Someone would have to log on to each computer in the domain as an administrator and grant the service account to have the Logon As A Service privilege. It is much easier to create a global group called DomainServices and grant the group the Logon As A Service privilege on each computer. This way, any account you later add to the group will inherit the privilege on each machine.*

When a user logs on, the operating system checks to see what privileges the user has been granted. A list is generated, and that list is then associated with the user's logon session. This list is only generated during logon; it is not updated through her logon session. Therefore, if her privileges change while she is logged on, she will have to log off and then log on again for the privileges to take place.

When the user attempts to perform some action that requires a specific privilege, the operating system will check the list associated with her logon session to see if she is allowed to perform the action. If the particular privilege is on this list, she is allowed to continue. Otherwise, she is prevented. This can be tracked in the Win32 Event Log using Win32 auditing.

The list of privileges that `Win32::Lanman` recognizes is in [Table 11.14](#). You can use the constant name of a privilege whenever you are referring to it. The constant name simply maps to a string called the *display name*. The display name (listed in [Table 11.15](#)) is not much better at describing the privilege than the constant name. Regardless, you can use either when referring to a privilege. However, if you refer to the display name, make sure you refer to it as a string, not as a constant.

## **Granting and Revoking Privileges**

The `GrantPrivilegeToAccount()` and `RevokePrivilegeFromAccount()` functions manage these privileges:

```
Win32::Lanman::GrantPrivilegeToAccount( $Server, $Privilege,  
 \@Accounts );  
Win32::Lanman::RevokePrivilegeFromAccount( $Server, $Privilege,  
 \@Accounts );
```

The first parameter (`$Server`) is the machine that is to grant or revoke the privilege. If this is `undef` or an empty string, the local machine is used.

The second parameter (`$Privilege`) determines which privilege is to be granted or revoked. This can be any one constant from [Table 11.14](#), or it can be a privilege display name string from [Table 11.15](#).

The third parameter (`\@Accounts`) is a reference to an array. This array contains a list of user accounts that are to either be granted or revoked the specified privilege. This can be an account from the machine specified by the first parameter (`$Server`), or it can be a domain account specified by the format `DomainName\UserName`.

If the functions are successful, they return `trUE (1)`; otherwise, they return `FALSE (0)`. The function will fail when it encounters the first error. Therefore, it might have successfully granted or revoked privileges to some users but not others. In this case, it still returns a `FALSE` value indicating failure. The script running these functions requires administrator privileges.

## Warning

*If you remove all rights from an account, the `RevokePrivilegeFromAccount()` function will also attempt to remove the user account.*

**Table 11.14. User Privileges**

Contant Name	Description
<code>SE_ASSIGNPRIMARYTOKEN_NAME</code>	Required to assign the primary token of a process.  <i>Replace a process-level token.</i>
<code>SE_AUDIT_NAME</code>	Required to generate audit-log entries. Give this privilege to secure servers.  <i>Generate security audits.</i>
<code>SE_BACKUP_NAME</code>	Required to perform backup operations.  <i>Back up files and directories.</i>
<code>SE_BATCH_LOGON_NAME</code>	Required to log on to a computer to run a batch job.  <i>Log on as a batch job.</i>
<code>SE_CHANGE_NOTIFY_NAME</code>	Required to receive notifications of changes to files or directories. This privilege also causes the system to skip all traversal access checks. It is enabled by default for all users.  <i>Bypass traverse checking.</i>
<code>SE_CREATE_PAGEFILE_NAME</code>	Required to create a paging file.  <i>Create a pagefile.</i>
<code>SE_CREATE_PERMANENT_NAME</code>	Required to create a permanent object.  <i>Create permanent shared objects.</i>
<code>SE_CREATE_TOKEN_NAME</code>	Required to create a primary token.  <i>Create a token object.</i>
<code>SE_DEBUG_NAME</code>	Required to debug a process.  <i>Debug programs.</i>

**Table 11.14. User Privileges**

Contant Name	Description
SE_INC_BASE_PRIORITY_NAME	Required to increase the base priority of a process.  <i>Increase scheduling priority.</i>
SE_INCREASE_QUOTA_NAME	Required to increase the quota assigned to a process.  <i>Increase quotas.</i>
SE_INTERACTIVE_LOGON_NAME	Required to interactively log on to the machine.  <i>Log on locally.</i>
SE_LOAD_DRIVER_NAME	Required to load or unload a device driver.  <i>Load and unload device drivers.</i>
SE_LOCK_MEMORY_NAME	Required to lock physical pages in memory.  <i>Lock pages in memory.</i>
SE_MACHINE_ACCOUNT_NAME	Required to create a machine account.  <i>Add workstations to domain.</i>
SE_NETWORK_LOGON_NAME	Required to log on to a machine over a network.  <i>Access this computer from the network.</i>
SE_PROF_SINGLE_PROCESS_NAME	Required to gather profiling information for a single process.  <i>Profile single process.</i>
SE_REMOTE_SHUTDOWN_NAME	Required to shut down a system using a network request.  <i>Force shutdown from a remote system.</i>
SE_RESTORE_NAME	Required to perform restore operations. This privilege enables you to set any valid user or group SID as the owner of an object.  <i>Restore files and directories.</i>
SE_SECURITY_NAME	Required to perform a number of security-related functions such as controlling and viewing audit messages. This privilege identifies its holder as a security operator.  <i>Manage auditing and security log.</i>
SE_SHUTDOWN_NAME	Required to shut down a local system.  <i>Shut down the system.</i>
SE_SYSTEM_ENVIRONMENT_NAME	Required to modify the nonvolatile RAM of systems that use

**Table 11.14. User Privileges**

Contant Name	Description
	this type of memory to store configuration information.
	<i>Modify firmware environment values.</i>
SE_SYSTEM_PROFILE_NAME	Required to gather profiling information for the entire system.
	<i>Profile system performance.</i>
SE_SYSTEMTIME_NAME	Required to modify the system time.
	<i>Change the system time.</i>
SE_TAKE_OWNERSHIP_NAME	Required to take ownership of an object without being granted discretionary access. This privilege allows the owner value to be set only to those values that the holder might legitimately assign as the owner of an object.
	<i>Take ownership of files or other objects.</i>
SE_TCB_NAME	This privilege identifies its holder as part of the trusted computer base. Some trusted protected subsystems are granted this privilege. This privilege is required to call the Win32::AdminMisc::LogonAsUser() function.
	<i>Act as part of the operating system.</i>
SE_UNSOLICITED_INPUT_NAME	Required to read unsolicited input from a terminal device.

**Table 11.15. User Privilege Display Names**

Display Name	Constant Name (From Table 11.14)
"SeAssignPrimaryTokenPrivilege"	SE_ASSIGNPRIMARYTOKEN_NAME
"SeAuditPrivilege"	SE_AUDIT_NAME
"SeBackupPrivilege"	SE_BACKUP_NAME
"SeBatchLogonRight"	SE_BATCH_LOGON_NAME
"SeChangeNotifyPrivilege"	SE_CHANGE_NOTIFY_NAME
"SeCreatePagefilePrivilege"	SE_CREATE_PAGEFILE_NAME
"SeCreatePermanentPrivilege"	SE_CREATE_PERMANENT_NAME
"SeCreateTokenPrivilege"	SE_CREATE_TOKEN_NAME
"SeDebugPrivilege"	SE_DEBUG_NAME
"SeIncreaseBasePriorityPrivilege"	SE_INC_BASE_PRIORITY_NAME
"SeIncreaseQuotaPrivilege"	SE_INCREASE_QUOTA_NAME
"SeInteractiveLogonRight"	SE_INTERACTIVE_LOGON_NAME

**Table 11.15. User Privilege Display Names**

Display Name	Constant Name (From Table 11.14)
"SeLoadDriverPrivilege"	SE_LOAD_DRIVER_NAME
"SeLockMemoryPrivilege"	SE_LOCK_MEMORY_NAME
"SeMachineAccountPrivilege"	SE_MACHINE_ACCOUNT_NAME
"SeNetworkLogonRight"	SE_NETWORK_LOGON_NAME
"SeProfileSingleProcessPrivilege"	SE_PROF_SINGLE_PROCESS_NAME
"SeRemoteShutdownPrivilege"	SE_REMOTE_SHUTDOWN_NAME
"SeRestorePrivilege"	SE_RESTORE_NAME
"SeSecurityPrivilege"	SE_SECURITY_NAME
"SeServiceLogonRight"	SE_SERVICE_LOGON_NAME
"SeShutdownPrivilege"	SE_SHUTDOWN_NAME
"SeSystemEnvironmentPrivilege"	SE_SYSTEM_ENVIRONMENT_NAME
"SeSystemProvilePrivilege"	SE_SYSTEM_PROFILE_NAME
"SeSystemtimePrivilege"	SE_SYSTEMTIME_NAME
"SeTakeOwnershipPrivilege"	SE_TAKE_OWNERSHIP_NAME
"SeTcbPrivilege"	SE_TCB_NAME
"SeUnsolicitedInputPrivilege"	SE_UNSOLICITED_INPUT_NAME

### **Enumerating User Privileges**

You can discover what privileges are assigned to a particular user or group by calling the `EnumAccountPrivileges()` function:

```
Win32::Lanman::EnumAccountPrivileges( $Server, $Account,
\@Privileges )
```

The first parameter (`$Server`) is the machine that is to grant or revoke the privilege. If this is `undef` or an empty string, the local machine is used.

The second parameter (`$Accounts`) is the user account. This can be an account from the machine specified by the first parameter (`$Server`), or it can be a domain account specified by the format `DomainName\UserName`.

The third parameter (`\@Privileges`) is a reference to an array. The array will be populated with all of the privileges assigned to the particular user. If the function is successful, it returns `trUE (1)`; otherwise, it returns `FALSE (0)`. If the user is not granted any privileges, the function fails, and a call to `Win32::Lanman::GetLastError()` returns a value of `2`.

### **Tip**

An account can be granted a privilege through membership in a group. In this case, the `Win32::Lanman::EnumAccountPrivilege()` function will not report the privilege if the username is specified. You would have to call this function once for each group of which the user is a member to collect the full set of privileges assigned to the user. This can be done using the `Win32::Lanman::NetUserGetGroups()` function.

## **Enumerating Users Who Have A Privilege**

You can discover the list of user accounts that have been granted a particular privilege by calling the `EnumPrivilegeAccounts()` function:

```
Win32::Lanman::EnumPrivilegeAccounts( $Server, $Privilege,
                                         \@Accounts )
```

The first parameter (`$Server`) is the machine that is to grant or revoke the privilege. If this is `undef` or an empty string, the local machine is used.

The second parameter (`$Privilege`) determines which privilege is being considered. This can be any one constant from [Table 11.14](#), or it can be a privilege display name string from [Table 11.15](#).

The third parameter (`\@Accounts`) is a reference to an array. This array will be populated with a list of user accounts that have been granted the specified privilege.

If the function is successful, it returns `true (1)`; otherwise, it returns `false (0)`. If no users are granted the specified privilege, the function fails, and a call to `Win32::Lanman::GetLastError()` returns a value of 259.

## **Case Study**

Jane is a network administrator for a large company. Lately, she has been running into situations in which she has to modify privileges for various accounts on her servers. The server farm that she administers runs many different services, each with its own unique needs. For example, the tape backup service must be granted the privileges to back up and restore files (thus bypassing file security).

She uses the script in [Example 11.22](#) to perform the task of granting and revoking user account privileges. The script also adds the capability to discover what accounts are granted a particular privilege as well as what privileges are granted to what accounts.

This script was taken from my book *Win32 Perl Scripting: The Administrator's Handbook* (New Riders).

### **Example 11.22 Managing user privileges**

```
01. use Win32::API;
02. use Win32::AdminMisc;
03. use Win32::Lanman;
04.
05. @PRIVILEGES = qw(
```

```

06.     SE_CREATE_TOKEN_NAME
07.     SE_ASSIGNPRIMARYTOKEN_NAME
08.     SE_LOCK_MEMORY_NAME
09.     SE_INCREASE_QUOTA_NAME
10.     SE_UNSOLICITED_INPUT_NAME
11.     SE_MACHINE_ACCOUNT_NAME
12.     SE_TCB_NAME
13.     SE_SECURITY_NAME
14.     SE_TAKE_OWNERSHIP_NAME
15.     SE_LOAD_DRIVER_NAME
16.     SE_SYSTEM_PROFILE_NAME
17.     SE_SYSTEMTIME_NAME
18.     SE_PROF_SINGLE_PROCESS_NAME
19.     SE_INC_BASE_PRIORITY_NAME
20.     SE_CREATE_PAGEFILE_NAME
21.     SE_CREATE_PERMANENT_NAME
22.     SE_BACKUP_NAME
23.     SE_RESTORE_NAME
24.     SE_SHUTDOWN_NAME
25.     SE_DEBUG_NAME
26.     SE_AUDIT_NAME
27.     SE_SYSTEM_ENVIRONMENT_NAME
28.     SE_CHANGE_NOTIFY_NAME
29.     SE_REMOTE_SHUTDOWN_NAME
30.     SE_INTERACTIVE_LOGON_NAME
31.     SE_NETWORK_LOGON_NAME
32.     SE_BATCH_LOGON_NAME
33.     SE_SERVICE_LOGON_NAME
34. );
35.
36. $LPDN = new Win32::API( 'advapi32.dll',
37.                         'LookupPrivilegeDisplayName',
38.                         [P,P,P,P,P], I )
39.         || die "Unable to locate the
LookupPrivilegeDisplayName().\n";
40. foreach my $Privilege ( @PRIVILEGES )
41. {
42.     my $Size = 256;
43.     my $szDisplayName = "\x00" x $Size;
44.     my $dwSize = pack( "L", $Size );
45.     my $dwLangId = pack( "L", 0 );
46.     my $PrivString = eval "$Privilege";
47.     $LPDN->Call( $Config{machine}, $PrivString,
48.                   $szDisplayName, $dwSize, $dwLangId );
49.     $szDisplayName =~ s/\x00//g;
50.     $PRIVILEGES{$Privilege} = {
51.         comment => $szDisplayName,
52.         display => $PrivString,
53.         name => $Privilege,
54.     };
55.     $PRIVILEGE_VALUES{uc $PrivString} = $Privilege;
56. }

```

```

57.
58. Configure( \%Config, @ARGV );
59. if( $Config{help} )
60. {
61.     Syntax();
62.     exit( 0 );
63. }
64.
65. if( "" ne $$Config{domain} )
66. {
67.     Win32::Lanman::NetGetDCName( '',
68.                                     $Config{domain},
69.                                     \$Config{machine} );
70. }
71. elsif( "" eq $$Config{machine} )
72. {
73.     Win32::Lanman::NetGetDCName( '',
74.                                     Win32::DomainName(),
75.                                     \$Config{machine} );
76. }
77.
78. if( $Config{display_privileges} )
79. {
80.     my @PrivList;
81.
82.     # Display all privileges
83.     if( scalar @{$Config{items}} )
84.     {
85.         foreach my $Priv ( @{$Config{items}} )
86.         {
87.             $Priv = MatchPrivilege( $Priv ) || next;
88.             push( @PrivList, $Priv );
89.         }
90.     }
91.     else
92.     {
93.         push( @PrivList, sort( keys( %PRIVILEGES ) ) );
94.     }
95.     foreach $Key ( @PrivList )
96.     {
97.         print "$Key:\n";
98.         print "\tDisplay name: $PRIVILEGES{$Key}->{display}\n";
99.         print "\tComment: $PRIVILEGES{$Key}->{comment}\n";
100.        if( "" ne $$PRIVILEGES{$Key}->{comment} );
101.        print "\n";
102.    }
103. }
104. elsif( $Config{user_rights} )
105. {
106.     # Display who has been enabled for a specific privilege
107.     foreach $Privilege ( @{$Config{items}} )
108.     {

```

```

109.    my @SidList;
110.    my $PrivKey = uc MatchPrivilege( $Privilege ) || next;
111.
112.    print "$PrivKey ($PRIVILEGES{$PrivKey}->{display}):\\n";
113.    if( Win32::Lanman::LsaEnumerateAccountsWithUserRight(
114.                                              $Config{machine},
115.                                              $PRIVILEGES{$PrivKey}-
116.                                              >{display},
117.                                              \@SidList ) )
118.    {
119.        my @SidData;
120.        Win32::Lanman::LsaLookupSids( $Config{machine},
121.                                         \@SidList,
122.                                         \@SidData );
123.        foreach my $Data ( @SidData )
124.        {
125.            print "\t", (("" ne $Data->{domain})? "$Data-
126.            >{domain}\\" : " " );
127.            print "$Data->{name}\\n";
128.        }
129.    }
130. }
131. else
132. {
133.     # Display what privilege has been enabled for specific
accounts
134.     my @AccountList;
135.     my @AccountInfo;
136.     my %TempAccountList;
137.
138.     # Expand any wildcards in the user groups...
139.     foreach my $Account ( @{$Config{items}} )
140.     {
141.         if( $Account =~ /\*$/ )
142.         {
143.             my( $Prefix ) = ( $Account =~ /^(.*)\*$/ );
144.             my @Accounts;
145.             Win32::AdminMisc::GetUsers( $Config{machine},
146.                                         $Prefix,
147.                                         \@Accounts );
148.             map
149.             {
150.                 $TempAccountList{lc $_} = $_;
151.             } @Accounts;
152.         }
153.     else
154.     {
155.         $TempAccountList{uc $Account} = $Account;
156.     }
157. }

```

```

158.  # Create a non-duplicate list of user accounts from the
temp hash
159. foreach my $Key ( sort( keys( %TempAccountList ) ) )
160. {
161.     push( @AccountList, $TempAccountList{$Key} );
162. }
163.
164. if( scalar @{$Config{add_privileges}} )
165.     || scalar @{$Config{remove_privileges}} )
166. {
167.     my @SidList;
168.     Win32::Lanman::LsaLookupNames( $Config{machine},
169.                                     \@AccountList,
170.                                     \@SidList );
171.     foreach my $Sid ( @SidList )
172.     {
173.         if( scalar @{$Config{add_privileges}} )
174.         {
175.             Win32::Lanman::LsaAddAccountRights( $Config{machine},
176.                                                 $Sid->{sid},
177.
$Config{add_privileges} );
178.         }
179.         if( scalar @{$Config{remove_privileges}} )
180.         {
181.
Win32::Lanman::LsaRemoveAccountRights( $Config{machine},
182.                                         $Sid->{sid},
183.
$Config{remove_privileges},
184.                                         $Config{remove_all} );
185.         }
186.     }
187. }
188. ReportAccountPrivileges( @AccountList );
189. }
190.
191. sub ReportAccountPrivileges
192. {
193.     my( @AccountList ) = @_;
194.
195.     $~ = PrivilegeDump;
196.     Win32::Lanman::LsaLookupNames( $Config{machine},
197.                                     \@AccountList,
198.                                     \@AccountInfo );
199.     for( $Index = 0; $Index < scalar @AccountInfo; $Index++ )
200.     {
201.         my @Rights;
202.         my $Account = $AccountInfo[$Index];
203.         $Account->{name} = $AccountList[$Index];
204.         print "$Account->{domain}\\\" if( \" " ne $$Account-
>{domain} );

```

```

205.     print "$Account->{name}";
206.
207.     # Check that the account exists
208.     if( 8 > $Account->{use} )
209.     {
210.         print ":\n";
211.
212.         if( Win32::Lanman::LsaEnumerateAccountRights( $Config{machine},
213.                                                       $Account-
214.                                                       >{sid},
215.                                                       \@Rights ) )
216.             {
217.                 map
218.                     {
219.                         $Priv{name} = $PRIVILEGE_VALUES{uc $_};
220.                         $Priv{display} = $_;
221.                         $Priv{comment} = $PRIVILEGES{$Priv{name}}-
222.                           >{comment};
223.                         write;
224.                     } @Rights;
225.             }
226.         else
227.             {
228.                 print "... account does not exist.\n";
229.             }
230.         print "\n";
231.     }
232.
233. sub MatchPrivilege
234. {
235.     my( $PrivRoot ) = uc shift @_;
236.     my $PrivKey = $PrivRoot;
237.     # Is the privilege a valid display name privilege?
238.     if( ! defined ( $PrivKey = $PRIVILEGE_VALUES{$PrivKey} ) )
239.     {
240.         # Is the privilege a normal privilege?
241.         $PrivKey = $PrivRoot;
242.         if( ! defined ( $PrivKey = $PRIVILEGES{$PrivKey}-
243.                       >{name} ) )
244.             {
245.                 # In case the user entered only the base name of the
246.                 # privilege
247.                 $PrivKey = "SE_" . $PrivRoot . "_NAME";
248.                 if( ! defined ( $PrivKey = $PRIVILEGES{uc $PrivKey}-
249.                               >{name} ) )
250.                     {
251.                         # In case the user entered only the base of the
252.                         # display name

```

```

249.         $PrivKey = "Se" . $PrivRoot . "Privilege";
250.         if( ! defined ( $PrivKey = $PRIVILEGE_VALUES{uc
251.             $PrivKey} ) )
252.             {
253.                 # One last chance...
254.                 $PrivKey = "SE" . $PrivRoot . "Right";
255.                 $PrivKey = $PRIVILEGE_VALUES{uc $PrivKey};
256.             }
257.         }
258.     }
259.     return( $PrivKey );
260. }
261.
262. sub Configure
263. {
264.     my( $Config, @Args ) = @_;
265.     while( my $Arg = shift @Args )
266.     {
267.         my( $Prefix ) = ( $Arg =~ /(^[-\//])/ );
268.         if( "" ne $$Prefix )
269.         {
270.             $Arg =~ s#^[-/]##;
271.             if( "+" eq $$Prefix )
272.             {
273.                 # Adding a privilege
274.                 my $Priv = MatchPrivilege( $Arg ) || next;
275.                 push( @{$Config->{add_privileges}}, 
276.                     $PRIVILEGES{$Priv}->{display} );
277.             }
278.             elsif( $Arg =~ /p$/i )
279.             {
280.                 # Request to display all rights
281.                 $Config->{display_privileges} = 1;
282.             }
283.             elsif( $Arg =~ /s$/i )
284.             {
285.                 # Specified displaying user rights
286.                 $Config->{user_rights} = 1;
287.             }
288.             elsif( $Arg =~ /d$/i )
289.             {
290.                 # Specify a domain to create the account
291.                 $Config->{domain} = shift @Args;
292.             }
293.             elsif( $Arg =~ /m$/i )
294.             {
295.                 # Specify what machine the account lives on
296.                 $Config->{machine} = "\\\\" . shift @Args;
297.                 $Config->{machine} =~ s/^(\\\\)+/\\\\\/;
298.             }
299.             elsif( $Arg =~ /(h|help)/i )

```

```

300.          {
301.              # Request help
302.              $Config->{help} = 1;
303.          }
304.      else
305.      {
306.          if( "/" eq $Prefix )
307.          {
308.              # An unknown switch
309.              $Config->{help} = 1;
310.          }
311.      else
312.      {
313.          # We get here if the prefix was -
314.          # and no valid flag matched the switch
therefore...
315.          # Removing a privilege
316.          if( "*" eq $Arg )
317.          {
318.              $Config->{remove_all} = 1;
319.              # Push * onto the remove array. It will be
320.              # anyway since we will use the "remove all"
321.              # This way the script will see the array is not
322.              # and attempt to remove privileges.
323.              push( @{$Config->{remove_privileges}}, $Arg );
324.          }
325.          else
326.          {
327.              my $Priv = MatchPrivilege( $Arg )
328.                  || ($Config->{help} = 1);
329.              push( @{$Config->{remove_privileges}},
330.                  $PRIVILEGES{$Priv}->{display} );
331.          }
332.      }
333.  }
334. }
335. else
336. {
337.     push( @{$Config->{items}}, $Arg );
338. }
339. }
340. if( 0 == scalar @{$Config->{items}} && ! $Config-
>{display_privileges} )
341. {
342.     $Config->{help} = 1;
343. }
344. }
345.
346. sub Syntax

```

```

347. {
348.   my( $Script ) = ( $0 =~ /([^\\\]*?)$/ );
349.   my( $Line ) = "-" x length(( $Script ));
350.
351.   print <<EOT;
352.
353. $Script
354. $Line
355. Manages account privileges
356.
357. Syntax:
358.   perl $Script [-m Machine | -d Domain] -p
359.   perl $Script [-m Machine | -d Domain] -s Priv [Priv2 ...]
360.   perl $Script [-m Machine | -d Domain] [-|+Priv] Account
[Account2 ...]
361.           -m Machine...All accounts and privileges are resident
on the
362.                   specified machine.
363.           -d Domain...All accounts and privileges are resident
in the
364.                   specified domain.
365.           -s Priv.....Show all accounts that have been granted
the specified
366.                   privilege. Some accounts may not show if
they are
367.                   granted the privilege through a group
membership.
368.           -Priv.....Removes the privilege from the specified
accounts.
369.                   Specify as many of these switches as
necessary.
370.                   Specify * to remove ALL privileges.
371.           +Priv.....Adds the privilege to the specified
accounts.
372.                   Specify as many of these switches as
necessary.
373.           Account.....Show all privileges granted to this
specified account.
374.           If used in conjunction with the -Priv or
+Priv then the
375.                   privileges are assigned or removed first.
The resulting
376.                   privilege set is then displayed.
377.                   This account can end with a * char to
indicate
378.                   all accounts that begin with the
specified string.
379.           If no domain or machine is specified then the current
domain
380.                   is used.
381. EOT
382. }

```

```

383.
384. format PrivilegeDump =
385.     @<<<<<<<<<<<<<<<<<<
386.     $Priv{name},                      $Priv{comment}
387. ~
388.                                     $Priv{comment}
389. ~
390.                                     $Priv{comment}
391. ...

```

## Machine Policies

Every machine has policies that govern how security is managed. These policies can be queried and modified by the `LsaQueryInformationPolicy()` and `LsaSetInformationPolicy()` functions:

```

Win32::Lanman::LsaQueryInformationPolicy( $Server, $Policy,
\%Info );
Win32::Lanman::LsaSetInformationPolicy( $Server, $Policy, \%Info);

```

The first parameter (`$Server`) is the machine that is to grant or revoke the privilege. If this is `undef` or an empty string, the local machine is used.

The second parameter (`$Policy`) represents what policy is being considered. This can be any one value from [Table 11.16](#).

The third parameter (`\%Info`) is a reference to a hash. This hash is populated with policy information (for `LsaQueryInformationPolicy()`) or is used to modify policy (for `LsaSetInformationPolicy()`). The different keys and values of this hash depend on what policy value is passed as the second parameter.

If the functions are successful, they return `true (1)`; otherwise, they return `false (0)`.

[Example 11.23](#) shows how you query policy settings. In the example, line 18 calls the `LsaQueryInformationPolicy()` function and retrieves the event auditing configuration. Of all the policy configuration settings, this particular one is the most confusing because the `%Info` hash is populated with an anonymous array in the `eventauditingoptions` key.

This array consists of entries that describe what events are being logged and in what fashion. The index of each value indicates what event the value represents. [Table 11.19](#) lists all of the different event names. These constants are used as index numbers. So if you wanted to examine the event that represents a user logging on to the machine, you would refer to the following:

```
$Option = $EventArray[AuditCategoryAccountLogon];
```

In this case, the `$Option` variable would hold a value that indicates what is audited when a user logs on.

A particular array index value is a bitmask that can contain different options from [Table 11.20](#). These indicate whether the system will audit a successful event, a failed event, or no event.

[Example 11.24](#), on the other hand, will configure a machine to enable event logging. This is how a network administrator can configure remote machines to audit different events. Lines 8 through 21 create an array called `@Options`, which will represent all the different event auditing options to be set. The array initially is configured with every entry set to `POLICY_AUDIT_EVENT_UNCHANGED`. This is required for any event that is not being configured. Otherwise, the LSA subsystem would assume you meant to place a `0` value, thus overwriting any existing value for the event. This enables you to configure event auditing for only a couple events at a time.

Notice that lines 4 through 7 create the `%Info` hash. The `eventauditingoptions` key is set with a reference to the `@Options` array. This enables the script to later modify the `@Options` array and have those changes reflected in the hash as it is passed into the `LsaSetInformationPolicy()` function (line 28).

### **Example 11.23 Querying auditing policies on a machine**

```
01. use Win32::Lanman;
02. my @AUDIT_EVENTS;
03. my $Server = shift @ARGV || Win32::NodeName();
04. foreach my $Event ( qw(
05.         AuditCategorySystem
06.         AuditCategoryLogon
07.         AuditCategoryObjectAccess
08.         AuditCategoryPrivilegeUse
09.         AuditCategoryDetailedTracking
10.        AuditCategoryPolicyChange
11.        AuditCategoryAccountManagement
12.        AuditCategoryDirectoryServiceAccess
13.        AuditCategoryAccountLogon
14.    ) )
15. {
16.     $AUDIT_EVENTS[eval "&$Event"] = $Event;
17. }
18. if( Win32::Lanman::LsaQueryInformationPolicy( $Server,
19.
PolicyAuditEventsInformation,
20.                                         \%Info ) )
21. {
22.     my $State = (( $Info{auditingmode} )? "enabled": "disabled" );
23.     print "Auditing is $State on $Server.\n";
24.     if( $Info{auditingmode} )
25.     {
26.         my $iIndex = 0;
27.         print "The auditing settings are:\n";
28.         while( $iIndex < scalar @{$Info{eventauditingoptions}} )
29.         {
30.             my $Options = $Info{eventauditingoptions}->[$iIndex];
```

```

31.     print "\t$AUDIT_EVENTS[$iIndex]:";
32.     if( $Options & POLICY_AUDIT_EVENT_NONE || 0 == $Options )
33.     {
34.         print " not audited";
35.     }
36.     else
37.     {
38.         print " successful" if( $Options &
POLICY_AUDIT_EVENT_SUCCESS );
39.         print " failure" if( $Options &
POLICY_AUDIT_EVENT_FAILURE );
40.     }
41.     print "\n";
42.     $iIndex++;
43.   }
44. }
45. }
46. else
47. {
48.   print "Error querying auditing policy.\n";
49.   print "Error: ",
Win32::FormatMessage( Win32::Lanman::GetLastError() );
50. }

```

#### **Example 11.24 Configuring a machine to audit**

```

01. use Win32::Lanman;
02. my @Options;
03. my $Server = shift @ARGV || Win32::NodeName();
04. my %Info = (
05.   'auditingmode' => 1,
06.   'eventauditingoptions' => \@Options
07. );
08. foreach my $Event ( qw(
09.           AuditCategorySystem
10.          AuditCategoryLogon
11.          AuditCategoryObjectAccess
12.          AuditCategoryPrivilegeUse
13.          AuditCategoryDetailedTracking
14.          AuditCategoryPolicyChange
15.          AuditCategoryAccountManagement
16.          AuditCategoryDirectoryServiceAccess
17.          AuditCategoryAccountLogon
18.        ) )
19. {
20.   $Options[eval "&$Event"] = POLICY_AUDIT_EVENT_UNCHANGED;
21. }
22. $Options[AuditCategoryAccountLogon] =
POLICY_AUDIT_EVENT_SUCCESS
23. |
POLICY_AUDIT_EVENT_FAILURE;

```

```

24. $Options[AuditCategoryPolicyChange] =
POLICY_AUDIT_EVENT_SUCCESS
25. |
POLICY_AUDIT_EVENT_FAILURE;
26. $Options[AuditCategoryObjectAccess] =
POLICY_AUDIT_EVENT_FAILURE;
27. $Options[AuditCategoryLogon] = POLICY_AUDIT_EVENT_FAILURE;
28. if( Win32::Lanman::LsaSetInformationPolicy( $Server,
29.
PolicyAuditEventsInformation,
30.                                         \%Info ) )
31. {
32.   print "Configuration was successful!\n";
33. }
34. else
35. {
36.   print "Error querying auditing policy.\n";
37.   print "Error: ",
Win32::FormatMessage( Win32::Lanman::GetLastError() );
38. }

```

**Table 11.16. Machine Policies**

<b>Policy Constant</b>	<b>Description</b>
PolicyAccountDomainInformation	Query or set the name and SID of the machine's account domain.
PolicyAuditEventsInformation	Query or set the machine's auditing rules. You can enable or disable auditing and specify the types of events that are audited. Policy properties are listed in <a href="#">Table 11.18</a> .
PolicyAuditFullQueryInformation	This value is obsolete. Policy properties are listed in <a href="#">Table 11.21</a> .
PolicyAuditFullSetInformation	This value is obsolete. Policy properties are listed in <a href="#">Table 11.22</a> .
PolicyAuditLogInformation	Retrieves information on audit logs. This is obsolete and is not used. Policy properties are listed in <a href="#">Table 11.23</a> .
PolicyDefaultQuotaInformation	This value is obsolete. Policy properties are listed in <a href="#">Table 11.24</a> .
PolicyDnsDomainInformation	Query or set domain name system (DNS) information about the account domain associated with a policy object. Policy properties are listed in <a href="#">Table 11.25</a> .
PolicyLsaServerRoleInformation	This only applies to Windows 2000/XP.
PolicyPdAccountInformation	Query or set the role of an LSA server. Policy properties are listed in <a href="#">Table 11.26</a> .
PolicyPdAccountInformation	This value is obsolete. Policy properties are listed in <a href="#">Table 11.26</a> .

**Table 11.16. Machine Policies**

Policy Constant	Description
	<a href="#">11.27.</a>
PolicyPrimaryDomainInformation	Query or set the name and SID of the machine's primary domain. Policy properties are listed in <a href="#">Table 11.28</a> .
PolicyReplicaSourceInformation	This is obsolete for Windows 2000/XP. Instead use <a href="#">PolicyDnsDomainInformation</a> . This value is obsolete. Policy properties are listed in <a href="#">Table 11.29</a> .

**Table 11.17. DNS Domain Policy Information**

Key Name	Description
modifiedid	An integer that is incremented each time anything in the LSA database is modified. This value is modified only on primary domain controllers.

**Table 11.18. Audit Events Policy Information**

Key Name	Description
auditingmode	A Boolean value indicating whether auditing is enabled.
eventauditingoptions	This key's value is a reference to an array. Each index number of the array matches an index constant name in <a href="#">Table 11.19</a> . The index name describes the event that the array entry represents. Refer to these index names as a constant value.
	The value in each array entry indicates how the event should be handled. This value can be any combination of options (logically OR'ed together) from <a href="#">Table 11.20</a> . Note that some combinations of options are not supported. Refer to the table for details.
maximumauditeventcount	The maximum number of recognized events. This is a read-only property.

**Table 11.19. Audit Event Index Names**

Index Constant	Description
AuditCategorySystem	Audit attempts to shut down or restart the computer. Also, audit events that affect system security or the security log.
AuditCategoryLogon	Audit attempts to log on to or log off from the system. Also, audit attempts to make a network connection.

**Table 11.19. Audit Event Index Names**

<b>Index Constant</b>	<b>Description</b>
AuditCategoryObjectAccess	Audit attempts to access securable objects such as files.
AuditCategoryPrivilegeUse	Audit attempts to use any Windows NT/2000/XP user privileges.
AuditCategoryDetailedTracking	Audit events such as program activation, some forms of handle duplication, indirect access to an object, and process exit.
AuditCategoryPolicyChange	Audit attempts to change policy object rules.
AuditCategoryAccountManagement	Audit attempts to create, delete, or change user or group accounts. Also, audit password changes.
AuditCategoryDirectoryServiceAccess	Audit attempts to access the directory service.
AuditCategoryAccountLogon	Audit logon attempts by privileged accounts that log on to the domain controller. These audit events are generated when the Kerberos Key Distribution Center logs on to the domain controller and by MSV1_0 for Windows NT 4.0 style logons.

**Table 11.20. Audit Event Options**

<b>Index Constant</b>	<b>Description</b>
POLICY_AUDIT_EVENT_UNCHANGED	For set operations, specify this value to leave the current options unchanged. If you are modifying some event options but not others, specify this value for the array index values you are not modifying. If you specify this option, do not specify any others.
POLICY_AUDIT_EVENT_SUCCESS	Generate audit records for successful events of this type. This option can be logically OR'ed with POLICY_AUDIT_EVENT_FAILURE.
POLICY_AUDIT_EVENT_FAILURE	Generate audit records for failed attempts to cause an event of this type to occur. This option can be logically OR'ed with POLICY_AUDIT_EVENT_SUCCESS.
POLICY_AUDIT_EVENT_NONE	Do not generate audit records for events of this type. This overrides any other option specified.

**Table 11.21. Audit Full Query Policy Information**

<b>Key Name</b>	<b>Description</b>
logisfull	Indicates whether or not the audit log is full. This is a read-only value.

**Table 11.21. Audit Full Query Policy Information**

Key Name	Description
<code>shutdownonfull</code>	Indicates whether or not the machine should shut down when the audit log is full.  A value of 0 means it should not shutdown. A value of 1 means it should shutdown.  This is a read-only value. To modify this value, you need to use the <code>PolicyAuditFullSetInformation</code> policy.

**Table 11.22. Audit Full Set Policy Information**

Key Name	Description
<code>shutdownonfull</code>	Indicates whether or not the machine should shut down when the audit log is full.  A value of 0 means it should not shutdown. A value of 1 means it should shutdown.  This is a write-only value. To query the value, you need to use the <code>PolicyAuditFullQueryInformation</code> .

**Table 11.23. Audit Log Policy Information**

Key Name	Description
<code>auditlogpercentfull</code>	Indicates the percentage of the Win32 Audit Log currently being used. This is a read-only property.
<code>maximumlogsize</code>	Specifies the maximum size of the Audit Log in kilobytes. This is a read-only property.
<code>auditretentionperiod</code>	Indicates the length of time that audit records are to be retained. Audit records are discardable if their timestamp predates the current time minus the retention period.
<code>auditlogfullshutdowninprogress</code>	Indicates whether or not a system shutdown is being initiated due to the security Audit Log becoming full. This condition will only occur if the system is configured to shut down when the log becomes full.  After a shutdown has been initiated, this flag will be set to <code>TRUE</code> . If an administrator is able to correct the situation before the shutdown becomes irreversible, this flag will be reset to <code>FALSE</code> .

**Table 11.23. Audit Log Policy Information**

Key Name	Description
	This is a read-only value.
timetoshutdown	If the <code>AuditLogFullShutdownInProgress</code> flag is set, this field contains the time left before the shutdown becomes irreversible. This is a readonly value.
nextauditrecordid	The audit record identifier for the next available audit record. This is a read-only property.

**Table 11.24. Default Quota Policy Information**

Key Name	Description
<code>pagedpoollimit</code>	Specifies the amount of paged pool memory assigned to the user. The paged pool is an area of system memory (physical memory used by the operating system) for objects that can be written to disk when they are not being used.  The value set in this member is not enforced by the LSA; therefore, it should only be set to 0, which causes the default value to be used.
<code>nonpagedpoollimit</code>	Specifies the amount of nonpaged pool memory assigned to the user. The nonpaged pool is an area of system memory (physical memory used by the operating system) for objects that cannot be written to disk but must remain in physical memory as long as they are allocated.  The value set in this member is not enforced by the LSA; therefore, it should only be set to 0, which causes the default value to be used.
<code>minimumworkingsetsize</code>	Specifies the minimum set size assigned to the user. The "working set" of a process is the set of memory pages currently visible to the process in physical RAM memory. These pages are present in memory when the application is running and are available for an application to use without triggering a page fault.  The value set in this member is not enforced by the LSA; therefore, it should only be set to 0, which causes the default value to be used.
<code>maximumworkingsetsize</code>	Specifies the maximum set size assigned to the user.  The value set in this member is not enforced by the LSA; therefore, it should only be set to 0, which causes the default value to be used.
<code>pagefilelimit</code>	Specifies the maximum size (in bytes) of the paging file, which is a reserved space on disk that backs up committed physical memory on the computer.  The value set in this member is not enforced by the LSA; therefore, it

**Table 11.24. Default Quota Policy Information**

Key Name	Description
<code>timelimit</code>	should only be set to 0, which causes the default value to be used.
	Indicates the maximum amount of time the process can run.
	The value set in this member is not enforced by the LSA; therefore, it should only be set to 0, which causes the default value to be used.

**Table 11.25. DNS Domain Policy Information**

Key Name	Description
<code>name</code>	The name of the primary domain.
<code>dnsdomainname</code>	The DNS name of the primary domain.
<code>dnsforestrname</code>	The DNS forest name of the primary domain.
	This is the DNS name of the domain at the root of the enterprise.
<code>domainguid</code>	The GUID of the primary domain.
<code>sid</code>	The SID of the primary domain.

**Table 11.26. LSA Server Role Policy Information**

Key Name	Description
<code>serverrole</code>	The role that the server plays. This is one of the following constant values:  <code>PolicyServerRolePrimary</code>  <code>PolicyServerRoleBackup</code>

**Table 11.27. Primary Domain (PD) Account Policy Information**

Key Name	Description
<code>name</code>	The name of the primary domain. This property is typically empty. It is not determined whether such behavior is expected or is a bug in the operating system. This is a read-only property.

**Table 11.28. Primary Domain Policy Information**

Key Name	Description
----------	-------------

**Table 11.28. Primary Domain Policy Information**

Key Name	Description
name	The name of the primary domain.
sid	The SID of the primary domain. This is a binary SID.

**Table 11.29. Replica Source Policy Information**

Key Name	Description
replicasource	This is an undocumented property.
replicaaccountname	This is an undocumented property.

## Secure Storage

Oftentimes, a script needs to store information in a secure way. For example, a coder would not want to hard code a password used to log on to a remote machine into a script. This way, anyone could simply look at the file to discover the password. Instead, she would want to somehow encrypt the data and then store it somewhere that no one else could get to. This is what *private data objects* are used for.

Private data objects are stored in the Registry. A script decides in what key name the data will be stored. Make sure you make the key name unique enough that it isn't already used by another application.

These functions enable a script to securely store data to be retrieved later. Four types of data can be stored:

- **Local private data objects:** The data is only accessible from the machine that stored it. It can not be accessed from another machine. Local private data objects all have key names that begin with `L$`.
- **Global private data objects:** The data must be stored on a domain controller. Once created, it is replicated to all domain controllers on the domain. Therefore, the data is eventually accessible from any domain controller. To create this type of private data, specify a domain controller as the server to store the data and prefix the key name with `G$`.
- **Machine private data objects:** The data is only accessible by the operating system. The data is not replicated to other machines. Applications can write this type of private data, but only the operating system can read it. Machine private data objects' key names all begin with `M$`.
- **General private data objects:** An application can access the data remotely, and the data is not replicated to any other machine. There is no special key prefix required.

Private data objects are secured by two techniques. First, the data is encrypted. When you retrieve the data, the data is automatically decrypted for you. Second, the data is stored in a Registry key protected by permissions that grant only the creator of the key access. Therefore, nobody else can access the key.

Private data objects can be managed using the `LsaStorePrivateData()` and `LsaRetrievePrivateData()` functions:

```
Win32::Lanman::LsaRetrievePrivateData( $Server, $Key, \$Data );
Win32::Lanman::LsaStorePrivateData( $Server, $Key, $Data );
```

The first parameter (`$Server`) is the machine where the private data object is stored. If this is `undef` or an empty string, the local machine is used.

The second parameter (`$Key`) is the name of the private data object. Specialized types of these objects require a special prefix in this key name.

The third parameter (`$Data`) is either a scalar variable or a reference to a scalar variable, depending on the function used. The `LsaStorePrivateData()` function will store the contents of this variable into private storage. The `LsaRetrievePrivateData()` will set the value of this variable with the private data.

If successful, both of these functions return `TRUE (0)`; otherwise, they return `FALSE (0)`. See [Example 11.25](#) for a complete listing.

## Note

*As of version 1.09 of Win32::Lanman, the `LsaStorePrivateData()` function is limited to only storing text strings. To be more precise, a string that contains binary data with embedded NULL characters will not store correctly. Other scalar data (text strings, integers, floating point numbers) will store correctly.*

### **Example 11.25 Accessing private data objects**

```
01. use Win32::Lanman;
02. my $Key = Win32::GetFullPathName( $0 );
03. # Registry keys can not have backslashes so replace
04. # them with a dash...
05. $Key =~ s/[:\\]/-/g;
06. print "Storing a private data object...\n";
07. if( Win32::Lanman::LsaStorePrivateData( "", $Key, "Hello
world!" ) )
08. {
09.     my $Data;
10.    print "Successfully stored private data!\n";
11.
12.    if( Win32::Lanman::LsaRetrievePrivateData( "", $Key,
\$Data ) )
13.    {
14.        print "The private data was '$Data'\n";
15.    }
16.    else
17.    {
18.        print "Error: ", Error();
```

```

19.    }
20. }
21. else
22. {
23.     print "Error: ", Error();
24. }
25. # Now let's try again but this time using
26. # a Global Private Data Object...
27. my $Domain = Win32::DomainName();
28. my $DomainController;
29. if( Win32::Lanman::NetGetAnyDCName( "", $Domain,
30. \$DomainController ) )
31. {
32.     $Key = 'G$' . $Key;
33.     print "Storing a global private data object...\n";
34.     if( Win32::Lanman::LsaStorePrivateData( $DomainController,
35.                                             $Key,
36.                                             "Hello Domain!" ) )
37.     {
38.         my $Data;
39.         print "Successfully stored global private data!\n";
40.     }
41.     if( Win32::Lanman::LsaRetrievePrivateData( $DomainController,
42.                                               $Key,
43.                                               \$Data ) )
44.     {
45.         print "The global private data was '$Data'\n";
46.     }
47.     else
48.     {
49.         print "Error: ", Error();
50.     }
51. }
52.     print "Error: ", Error();
53. }
54. }
55. else
56. {
57.     print "Unable to locate a domain controller.\n";
58.     print Error();
59. }
60.
61. sub Error
62. {
63.
return( Win32::FormatMessage( Win32::Lanman::GetLastError() ) .
"\n" );
64. }

```

## Summary

With all the complexity that makes up the Win32 security system, it is amazing that Perl can keep up with it. But by leveraging the `Win32::Perms` and `Win32::Lanman` extensions, not only can you set permissions on files and directories, you can also modify all sorts of other secured objects and manage privileges and policies as well.

# Chapter 12. Common Mistakes and Troubleshooting

This chapter investigates the common traps and pitfalls that plague Win32 Perl users, in particular when using the Win32 extensions.

All programmers have to look out for certain perils and pitfalls inherent to a programming language. Perl is no exception to this phenomenon. With Win32 Perl being in a state of constant change, it is inevitable that there will be problems.

Volumes could be written regarding all the possible issues, caveats, and exceptions that a Win32 Perl programmer must consider when coding. Instead, this chapter covers some of the more common problems that have hit the newsgroups and email listservers. The chances are good that most coders have stumbled upon some, if not all, of these issues at one time or another. Hopefully, this brief overview will help shine the light on at least some of the problems you might be facing.

## General Win32-Specific Mistakes

One of the most obvious differences between the UNIX and Win32 platforms is file paths. Any UNIX user can tell you that there is only one root with many subdirectories. All hard drives, network shares, floppy drives, and any other external media storage devices are mounted as subdirectories under the root. Because Perl was originally designed to work under the UNIX platform, this is what Perl expects.

The Win32 world, however, uses drive letters to designate a storage device. This, of course, limits the number of possible drives to 26 (A: through Z:). This does not include network shares that are not mapped to a drive letter.

Another difference between the two platforms is that UNIX uses a forward slash (/) to delimit directories, whereas Win32 (and DOS for that matter) uses a backslash (\).

These little subtleties can make all the difference when using Perl on a Win32 machine—especially when porting UNIX scripts to a Windows box.

The most common problems that Win32 Perl users face relate to the following:

- File paths
- File permissions

## File Paths

UNIX variants and Win32 systems vary in the way that files and paths are structured. UNIX generally makes use of the slash character (/) to delimit a path, as in `/usr/bin/perl`.

Win32 operating systems, however, use the backslash to delimit filenames and pathnames. Additionally, Win32 uses a drive letter that identifies a partition on a hard drive. This characteristic has been inherited from the older days when Windows was based on DOS (even though Windows 95 is still based on DOS). An example of a Win32 path is `c:\perl\bin\perl.exe`.

Win32 Perl can handle either style when referring to file and directory paths. The code in [Example 12.1](#), for example, works exactly the same as the code in [Example 12.2](#).

### **Example 12.1 Using UNIX-style paths**

```
01. if( open( FILE, "< c:/temp/MyFile.txt" ) )
02. {
03.     print <FILE>;
04.     close( FILE );
05. }
```

### **Example 12.2 Using Win32-style paths**

```
01. if( open( FILE, "< c:\\temp\\MyFile.txt" ) )
02. {
03.     print <FILE>;
04.     close( FILE );
05. }
```

Many Win32 Perl coders routinely use the slash rather than the backslash when writing paths because it is easier to both write and read. Consider creating a string that represents the file `C:\TEMP\MISC.TXT`. To make a Perl string, you will have to either use single quotation marks, as in the following:

```
$File = 'c:\temp\misc.txt';
```

Or you would have to escape the backslashes with another backslash, like this:

```
$File = 'c:\\temp\\\\misc.txt';
```

If you were to use the backslash (regardless of the use of single or double quotation marks), you would be doomed to stay on a Win32 machine. You would run into errors if you tried to run the script on a UNIX box.

The capability to use forward or backslashes makes Win32 Perl rather powerful because you can run scripts developed for UNIX machines on a Win32 box; however, when Win32 extensions come into play, this consistency breaks apart.

Because delimiters can be expressed as either a slash (/) or a backslash (\), one might be encouraged to use slashes when referring to something like a proper computer name. Doing this could generate a runtime error or just never resolve a computer name because most extensions do not convert slashes into backslashes.

Consider the `Win32::NetAdmin::UserGetAttributes()` function, for example. If you pass in a server name of `\server`, it will function correctly, but if you pass in `//server`, it will fail to locate the proper network server. It is important to note which extensions do and do not allow the slash to replace the backslash delimiter. If you have an error in which a machine name is not recognized, it might be due to this problem.

One additional caveat that pertains to backslashes must be pointed out. When you specify a proper computer name such as `\ServerA` using single quotation marks, Perl will interpret the prefixed backslashes as a single escaped backslash. So, calling a function like

```
Win32::NetAdmin::GetUsers( '\\"ServerA', FILTER_NORMAL_ACCOUNT,  
@UserList );
```

will fail because, after Perl has processed the parameters, the extension would see the first parameter as `\ServerA`. Notice how the two backslashes were interpreted as an escaped backslash, resulting in only one backslash. This is a "gotcha" that has claimed the lives of many scripts.

## File Permissions

Normally, when someone creates a Perl script, he has permissions to access the `Perl.exe` program and its various library files. It is easy to forget that other user accounts might not have permission on all the files needed to successfully run the script.

Suppose an administrator has written a script that opens a Registry key. Assume also that this is executed as a logon script for each user during logon. If the user does not have read permissions on the `Win32::Registry` extension file (`REGISTRY.PLL` or `REGISTRY.DLL`), the script will break with a runtime error because it could not load the extension.

Suppose another user logs on and the script runs. This user has full access on all files in the Perl directory, so he loads the `Win32::Registry` extension, unlike the preceding user. This user does not have read access on the Registry key that will be opened, however. This time the script will run, but it will not be able to access the Registry key (it will fail to open the key).

Both of these problems occur because permissions were not set for a given user. The author never encountered these problems because he had the required permissions. It is quite important to consider permission issues, especially when writing scripts that will run under different environments or under different user accounts.

More information on permissions and how to manage them can be found in [Chapter 11](#), "Security."

## Windows Scripting

One of the exciting features of Win32 platforms is their support for scripting. Although such support is not new to computer science, it is fairly new to Microsoft's operating systems.

The scripting capability introduced by Win32 platforms is known as *Windows Script Hosting* (WSH). This technology enables any language to register a *scripting engine*. Such an engine is a COM server component (refer to [Chapter 5](#), "Automation," for details on COM servers). This enables any application to execute a script, regardless of language.

To better describe this, consider a word processor that uses WSH technology as its macro language. This means that a user could create macros in PerlScript, VBScript, or even JavaScript. As long as there is a WSH engine registered with the operating system for the language, the word processor could use it.

After you have installed PerlScript, any WSH script can be created using Perl. There is no difference between Perl and PerlScript coding, with the exception that there are a couple small things you need to be aware of. These are described in the "Scripting Issues" section later in this chapter.

## Registering PerlScript

When you install ActivePerl, you have the option of installing PerlScript. This simply means that the Perl-based WSH engine will be registered with Win32. However, if you are manually installing Perl or you installed it without the PerlScript option, the engine is not going to be registered. Therefore, Perl will not be a WSH option.

This is easily corrected. In the Perl\bin directory, there is a file called `Perlse.dll`. This is the PerlScript WSH engine. It is actually a COM server component; therefore, it must be registered the way all COM server components are: using `RegSvr32.exe`.

To register the Perl WSH engine, simply execute the following command:

```
regsvr32 perlse.dll /s
```

This will update the Registry with the appropriate information that will register PerlScript with the WSH subsystem. Notice that the command specifies the `/s` option. This just means it will register silently. If you don't specify this option, the registration will still occur, but a GUI dialog box will appear indicating that the component has been registered. By specifying this "silent option," you can automate registering this component from a script. This is really useful when you automate installing Perl on many machines.

You can register a component such as this as many times as you like. Don't worry about calling this function over and over.

If for any reason you want to deregister the PerlScript WSH engine, simply add the `/u` option to specify that you want to uninstall the component:

```
regsvr32 perlse.dll /u /s
```

Again, you can leave off the `/s` option if you don't want to uninstall the component silently.

## Scripting Issues

Previously, I mentioned that there is no real difference between Perl and PerlScript. This is absolutely true with just a couple of exceptions.

The only real differences that a Perl coder needs to be aware of are as follows:

- Printing to `STDOUT` won't go where you think. You need to print to an appropriate output object. Any good book on Windows Shell Host scripting (such as Tim Hill's *Windows NT Shell Scripting*, published by New Riders) can explain how to print.
- There are usually some global COM objects defined that the script can access. There is no need to load the `Win32::OLE` extension; access to the COM object is automatically handled. Your script accesses the COM objects as if they were obtained using `Win32::OLE`.

## ASP Pages

Microsoft's Active Server Pages (ASP), which are so popular on the Web, make use of this WSH technology. By default, the Internet Information Server (IIS) web server expects Visual Basic Script to be used for ASP pages. This is totally configurable on a page-by-page basis and can include PerlScript instead of VBScript.

To specify that an ASP page will use PerlScript, you need to add the following line to the very beginning of your ASP page:

```
<%@ Language=PerlScript %>
```

This tells the ASP system to use PerlScript for the duration of the processing of the ASP page.

### **Predefined Objects**

ASP pages always define certain default COM objects (refer to [Chapter 5](#) for a full discussion on using COM in Perl and PerlScript). These COM objects are as follows:

- **\$Response:** Defines methods and properties for responding to the client's request for content
- **\$Request:** Defines methods and properties for obtaining information regarding the client's request for content
- **\$Session:** Defines methods and properties for discovering information about the client's interaction with the server
- **\$Server:** Defines methods and properties for interacting with the web server

Newer versions of ASP also define additional components. Check with your IIS and ASP documentation to discover what methods, properties, and objects are available.

### **Accessing Predefined Objects**

By far, the easiest mistake to make when using PerlScript in ASP pages is to forget that the predefined COM objects are stored in the main namespace. Simply stated, this means that when your script refers to an object such as `$Request`, it is really accessing `$main::Request`.

This is important to note is because, if you create Perl packages or modules that process ASP requests, you might fall into a trap. The trap is that you might try to access `$main::Request` from within your own package using the simple form `$Request`. The moment your code enters the scope of your module, the default namespace becomes the modules namespace.

For example, if you created a `Win32::Foo` module that looked like the one in [Example 12.3](#), everything would work just fine. However, if line 4 was as follows:

```
return( $Request->Cookies() );
```

The `GetCookie()` function would fail. This is because, when you enter this function, the *default namespace* has become `Win32::Foo`. Therefore, `$Request` is really the same as `$Win32::Foo::Request`. And because `$Win32::Foo::Request` has not been defined by the package, it would not return what the calling script expects.

### ***Example 12.3 Calling ASP default objects from a module***

```
01. package Win32::Foo;
02. sub GetCookie
03. {
04.     return( $main::Request->Cookies() );
05. }
06. 1;
```

It is best to always refer to the default ASP COM objects with their full namespace. That way, your script will never become confused and errors won't creep in.

#### **Tip:**

*It is important to always specify the full namespace of the default ASP COM objects. For example, always refer to the `$Request` object as `$main::Request`, although you can also refer to it as `$::Request`. This is shorthand for referring to the `main` namespace.*

*Anywhere you see a reference to `$main::Foo`, you can substitute the `main::` namespace delimiter with just `::`. This makes coding a tad easier.*

### ***Sending Output to a Browser***

When you run a CGI script using either `PerlIS.dll` or `perl.exe`, you can print to `STDOUT`, and that is sent directly to the client (such as a web browser). PerlScript, however, is not quite so understanding.

In PerlScript, an object called `$Response` has specific capabilities. How to use this object (and its brethren such as `$Server` and `$Session`) is beyond the scope of this book, but it does have one method worth examining here: the `Write()` method.

When your script would normally print to `STDOUT`, you just need to pass the outputted data to the `$Response->Write()` method, as in [Example 12.4](#) (line 6). Notice that instead of printing the scalar `$HTML`, we are sending it as a parameter into `$Response->Write()`.

### **Example 12.4 Printing in an ASP page using PerlScript**

```
01. <%@ Language=PerlScript %>
02. <%
03.     my $HTML = "<HTML><HEAD><TITLE>My Test</TITLE></HEAD><BODY> ";
04.     $HTML .= "<H1>This is my Test!!!</H1>";
05.     $HTML .= "</BODY></HTML> ";
06.     $::Response->Write( $HTML );
07.     $::Response->Flush();
08. %>
```

### **Write() and BinaryWrite()**

The ASP `$Response` object has two methods for sending information to the client browser: the `Write()` and `BinaryWrite()` methods. Basically, if you want to send any data (such as HTML code) to the client browser, you call the `Write()` method, passing in the string of data to print out anything (see [Example 12.5](#)).

### **Example 12.5 Using the `$Response->Write()` method**

```
01. <%@ Language=PerlScript %>
02. <HTML>
03. <HEAD>
04. <TITLE>Test</TITLE>
05. </HEAD>
06. <BODY>
07.
08. The current time on the server is:
09. <%
10.    my @Date = localtime();
11.    $Time = sprintf( "%02d:%02d:%02d",
reverse( (@Date)[0..2] ) );
12.    $::Response->Write( $Time );
13. %>
14. </BODY>
15. </HTML>
```

Interestingly enough, there is a shortcut to calling the `Write()` method using the `<%= xxx %>` format. In this case, you can embed this seamlessly in some HTML code, as in [Example 12.6](#). Notice that the `$Time` variable is determined early on in the script and then it is printed out with the `<%= $Time %>` tag (line 14). You can also toss a piece of code into the tag, as in line 15.

PerlScript converts these tags into code that calls the `Write()` method. For example, line 14 is converted into the following:

```
<%
$::Response->Write( $Time );
```

```
%>
```

Line 15 is converted into the following:

```
<%
    $::Response->Write( sprintf( "%02d:%02d:%02d" ,
reverse( (@Date)[3..5] ) ) );
%>
```

### **Example 12.6 Using the <%= %> tag**

```
01. <%@ Language=PerlScript %>
02. <%
03.     my @Date = localtime();
04.     # Modify the date array for the proper year and month
05.     $Date[5] += 1900;
06.     $Date[4] += 1;
07.     my $Time = sprintf( "%02d:%02d:%02d" ,
reverse( (localtime())[0..2] ) );
08. >
09. <HTML>
10.   <HEAD>
11.     <TITLE>Test</TITLE>
12.   </HEAD>
13.   <BODY>
14.     The current time on the server is: <%= $Time %>
15.     The current date on the server is: <%= 
sprintf( "%02d:%02d:%02d" , reverse(
    (@Date)[3..5] ) ) %>
16.   </BODY>
17. </HTML>
```

The `$Response` object's `Write()` method enables you to dump text data to the client browser. However, if you need to send binary (nontext) data to the client, the `Write()` method won't work. This is what the `BinaryWrite()` method is all about.

One would expect that you would simply pass in a binary Perl string and it is sent. For example, if you wanted to open a binary file and send it to the client, you would read a block of data from the file and then call `BinaryWrite()`, passing in the block of binary data. However, if you try this, you will find that the resulting data that the browser receives will be twice as large.

This is odd indeed. It turns out that the actual prototype of the `BinaryWrite()` method expects a variant array (refer to [Chapter 5](#) for details of variants) of unsigned, 1-byte chars (variant type of `VT_UI1`). When you think about it, an array of chars is exactly what a C-based string is. Perl manages strings a bit differently than C, however, so by default, when PerlScript converts a Perl string into a variant, it will cast it as a UNICODE-based character string (a variant `BSTR`) as opposed to an array of unsigned bytes. This means that when you pass your Perl string into `BinaryWrite()`, it is automatically converted into a UNICODE string. This is very, very bad.

The way around this is to first create a variant array of unsigned characters (line 27) and then populate that array with the contents of the Perl string (line 29); each byte of the string is stored into a different array element. Finally, the variant is passed into the `BinaryWrite()` method (line 31).

### **Example 12.7 12.7: Downloading a binary file to a browser**

```
01. <%@ Language=PerlScript %>
02. <%
03. use Win32::OLE::Variant;
04. # Discover the requested file name. First try
05. # examining the form. If, however, the HTML file
06. # that called this does not have a form then
07. # it examines the query string for a parameter like:
08. #     http://myserver.com/download.asp?FileName=MyFile.doc
09. my $Query = $::Request->Form( "FileName" )
10.           || $::Request->QueryString( "FileName" );
11. if( defined $Query )
12. {
13.     my $FileName = $Query->Item();
14.     my $Path = "c:\\temp\\$FileName";
15.     if( open( FILE, "< $Path" ) )
16.     {
17.         my $Buffer;
18.         my $Size = ( stat( $Path ) )[7];
19.         binmode( FILE );
20.         # We don't want cache/proxies to cache this download!
21.         $::Response->CacheControl( 'Private' );
22.         $::Response->AddHeader( "Content-Disposition",
"attachment";
                filename=\"$FileName\" );
23.         $::Response->AddHeader( "Content-Size", $Size );
24.         while( read( FILE, $Buffer, 10240 ) )
25.         {
26.             # Create a variant array of unsigned chars
27.             my $Variant = Variant( VT_ARRAY | VT_UI1,
length( $Buffer ) );
28.             # Populate the array with the file data
29.             $Variant->Put( $Buffer );
30.             # Write out the data to the client as a binary stream
31.             $::Response->BinaryWrite( $Variant );
32.         }
33.         close( FILE );
34.         # Force the flushing of buffers and end
35.         # the processing of this ASP page
36.         $::Response->End();
37.         exit;
38.     }
39.     else
40.     {
41.         $Error = "Unable to open '$Path'. Error: $!";
42.     }
43. }
44. else
```

```

45.  {
46.      $Error = "No file was specified to download";
47.  }
48. %>
49. <HTML>
50.   <HEAD>
51.     <TITLE>Error Downloading File</TITLE>
52.   </HEAD>
53.   <BODY>
54.     <H1>Error downloading file</H1>
55.     <%= $Error %>
56.   </BODY>
57. </HTML>

```

## CGI Script Problems

Now that the media has hyped the World Wide Web, CGI scripting has become a full-time job for some folks. Perl is a wonderful language to write CGI scripts because of its excellent text management, socket, and file capabilities. Using Perl for CGI scripts on a Win32 web server, however, brings along with it an entire armada of problems.

The most common Win32 Perl CGI-related problems can be boiled down to just a few topics:

- **Using `PerlIS.dll`.** This is ActiveState's ISAPI application that can run Perl scripts with more efficiency than `perl.exe`. When using this, you need to handle scripts a little bit differently than if you were just using `perl.exe`.
- **Security issues.** Permissions and security settings of a web server can cause your scripts to behave erratically.
- **Network communications.** When running Perl scripts as a CGI script, some network communications are impaired.

### **PerlIS.dll**

The ActiveState version of Win32 Perl supports an ISAPI application called `PerlIS.dll`. The filter will only run on web servers that support ISAPI applications. Installing `PerlIS.dll` on a machine running a web server that supports ISAPI applications provides fast Perl script loading. This curious piece of software speeds up CGI script loading because, after the application is loaded, it remains in memory until the web server process terminates. This eliminates Perl's process load time—the time it takes for the system to load and execute the `perl.exe` program (loading `perl.exe`, locating its needed DLL files, and then loading them). Chances are, you won't really notice speed improvements unless you are running many scripts at the same time. It is important to note that `PerlIS.dll` will not make your Perl scripts run any faster; however, they will *begin* to execute more quickly.

Consider a web server that takes one second to load and run `perl.exe`. This time includes locating and loading all needed `.dll` files (such as `Perl.dll`, `Perlcore.dll`, `PerlCRT.dll`, and any extension `.dll` files). If your web server receives a request to run a Perl-based CGI script, it will take one second to load Perl, then it must compile the script, and then execution begins. Suppose 10 requests come in at the same time for a Perl-based CGI script. Collectively, it would take at least 10 seconds for Perl to load for all the requests. Because the 10 requests occur at the same time,

however, there will be a bit of additional time overhead because you will have all 10 processes attempting to start simultaneously. On slower machines, this could stretch from 10 to 20 seconds or even longer.

Using `PerlIS.dll` radically reduces this time burden because no new processes must be started. A thread is obtained from a pool of waiting threads, taking practically no time overhead. Because all DLL files are already loaded, the entire load time is minimized to the time it takes to load the Perl script itself. Web servers that receive many hits on Perl-based CGI scripts can indeed see dramatic speed improvements. Servers with few Perl script requests, however, will probably not see much improvement. In web servers such as Microsoft's Internet Information Server (IIS), different file extensions can be mapped to different executables. Many administrators associate CGI scripts that have a `.pl` extension to run `perl.exe` and scripts with extensions of `.cgi` (or something similar) to run using `PerlIS.dll`. This gives the webmaster flexibility to use either method for Perl script execution. When installing `PerlIS.dll`, however, ActiveState suggests using the `.plx` suffix with `PerlIS.dll`. Really, it does not matter which extension you use with which process—the choice is totally up to the webmaster. It might be wise, however, to avoid the `.pl` extension because any hacker can identify that you are using Perl. This could be an invitation for a hacker who is familiar with Perl.

## **Default Directories**

When you run a script executed by the `PerlIS.dll` ISAPI application, the script might not know what a current directory is. This is an odd concept to have to contend with because Win32 supports a current directory, as does Perl. Any script that has been written assuming that a current directory exists might fail. [Example 12.8](#) works when run from a command line or when run as a CGI script using `perl.exe`, but it could fail if run from a web server using `PerlIS.dll`.

### **Example 12.8 Using the current directory fails under `PerlIS.dll`**

```
01. $ | = 1;
02. print "Content-type: text/html\n\n";
03. open( FILE, "< ./data.txt" ) || die "Could not open the file
($!).\n";
04. print join( "<br>", <FILE> );
05. close( FILE );
```

This limitation is due to how older ISAPI web servers handle this situation (such as IIS 3.0). `PerlIS.dll` does not impose it. Some web servers have corrected this and provide a current directory even for `PerlIS.dll`-based scripts. If you are using IIS 4.0 and are experiencing this particular problem, you might be able to correct it by changing the `CreateCGIWithNewConsole` setting to `1` in the MetaBase. This setting will enable a CGI script to make use of current and default directories, as well as use backticks and the `system()` function to spawn processes. [Example 12.9](#) demonstrates a simple Perl script that sets the MetaBase's `CreateCGIWithNewConsole` setting to `1`. This example uses the `Win32::OLE` extension, so it will only work with either the core distribution's `libwin32` or with Perl 5.005. If you have ActiveState pre-5.005 build on your web server (such as Build 316) and you have `DCOM` installed on another machine with Perl 5.005, you can run the script from that other machine. In this case, you will have to specify which web server to modify.

## **Tip**

*IIS version 4.0 and higher uses a data repository called the MetaBase for storing its settings. This is the `METABASE.BIN` file usually located in the `C:\WINNT\SYSTEM32\INETSRV` directory. It is not commonly known why this method of storing settings information is used instead of the Registry. Regardless, there are a few ways to change settings in the MetaBase, one of which is a MetaBase editor (similar to the Registry editor `REGEDIT.EXE` and `REGEDT32.EXE`) that comes with the IIS 4.0 Resource Kit. Alternatively, you can change values in the MetaBase using a utility called `csrss` and a Visual Basic script (both come with IIS 4.0). The script is in the `\winnt\system32\inetsrv\adminsamples` directory and will enable you to modify MetaBase settings. To set the `CreateCGIWithNewConsole` setting to 1 you can use the following:*

```
csrss adsutil.vbs SET w3svc/CreateCGIWithNewConsole "1"
```

*You can use the Perl script in [Example 12.9](#) to set this property as well.*

### **Example 12.9 Setting the `CreateCGIWithNewConsole` property for an IIS web server**

```
01. use Win32::OLE;
02. # Pass in a server name (or address) as the first parameter
03. # into the script otherwise assume to use the local host.
04. my $Server = "LocalHost" unless ( $Server = $ARGV[0] );
05. # Get a web server object
06. my $Object = "IIS://$Server/w3svc";
07. my $WebServer = Win32::OLE->GetObject( $Object, 1 ) || die;
08. # Set the property
09. $WebServer->{CreateCGIWithNewConsole} = 1;
10. $WebServer->SetInfo();
```

To avoid this problem, a Perl script must use full path names when referencing files and directories. This makes it difficult to move scripts from one server to another because a web server's CGI script directory might be in a physically different location from another server.

Some web servers provide an environmental variable in the CGI environment that contains the value of the current directory for the running script. A script can make use of this to get around the lack of a current directory, as illustrated in [Example 12.10](#). [Example 12.10](#) uses the environmental variable `PATH_TRANSLATED`, which IIS provides and is the full path to the script file.

The `PerlIS.dll` extension defines an environmental variable `PERLXS`. This enables any script to check and see whether it is running under this extension.

### **Example 12.10 Using environmental variables to determine the current directory**

```
01. $ | = 1;
02. print "Content-type: text/html\n\n";
03.
04. # We will dump this file from the current directory
05. $File = "page.htm";
06.
07. if( $Dir = $ENV{'PATH_TRANSLATED'} )
08. {
```

```

09. # The environmental variable exists so we can strip off the
10. # file name and we now have our directory path.
11. $Dir =~ s/(\\[^\\]*?)$//;
12. print "<br>Using env var: dir='$Dir'<p>\n";
13. }
14. else
15. {
16. # If the environmental variable does not exist assume this
17. # is
18. # not using PerlIS.dll and we can use a current directory.
19. print "<br>Not using env var!<p>\n";
19. $Dir = ".";
20. }
21. open( FILE, "< $Dir/$File" ) || die "Could not open '$File'
($!).\n";
22. print join("<br>", <FILE> );
23. close( FILE );

```

## Threads

The `PerlIS.dll` ISAPI application works by assigning a thread for each script executed. Generally, this approach is fine and makes for efficient management. Some Win32 extensions are not thread safe, however. This is not always a problem, but it can become one. Thread issues do not arise when running from a command line, so it is difficult to find them when debugging your scripts. Instead, they usually come up when you are running multiple instances of a script at the same time, such as in a production environment (typically online for the world to see).

The problem arises when an extension is not written to be thread aware. This just means that the author of the extension made assumptions that only one script would be accessing memory and data at a time. The `Win32::NetAdmin` extension, for example, records any error it generates into an internal variable called `lastError`. If a script calls a `Win32::NetAdmin` function that fails, it needs to call the `Win32::NetAdmin::GetError()` function to retrieve the contents of the `lastError` variable (which is the Win32 error that the function returned). The extension's `lastError` variable, however, is global in scope. This is not thread friendly.

Consider two scripts, Script A and Script B. Script A calls the `Win32::NetAdmin::GetUsers()` function to get the list of users. This function fails (because the specified server is offline, for example) and sets the `lastError` variable to `53 ("The network path was not found.")`. The script now has to figure out whether the function failed, and if it did, the script calls the `Win32::NetAdmin::GetError()` function to find out what the error number was. While Script A is figuring out whether the function failed, Script B makes a call into another `NetAdmin` function (checking whether a user is a member of a group), which is successful. This causes `lastError` to be set to the result indicating that the last function was successful (a value of `0`). By now, Script A has concluded that its attempt to get a list of users failed, so it will call the `GetError()` function. The result, however, will be the result of Script B's last call (the successful one).

This is known as a *race condition*, in which two processes fight for one resource. When the `PerlIS.dll` application is loaded, all scripts that it runs are executed using a different thread, but they all share the same memory. For those not familiar with threads, DLLs, and C coding, think of this as a problem of writing a big Perl script that uses a global variable that each function changes.

If one function changes the global variable, another function might not know that the variable has changed, so it assumes all is fine.

Perl was not designed to be thread safe, and therefore many of its extensions do not bother to worry about such issues. Now that the `PerlIS.dll` application has become quite popular and future versions of Perl will support threading, Perl extensions have had to change how they manage these issues. Consider a web site in which several administrators use CGI scripts to manage user accounts. If they are running these scripts at the same time, several threads could be overwriting the `lastError` variable. There would most likely be several conflicts.

A solution to this problem is either to guarantee that the scripts run one at a time by using a semaphore or some other mechanism or to configure the web server to only run sensitive scripts by using `perl.exe` rather than `perlIS.dll`. When threading is finally stable in Win32 Perl, it is inevitable that the authors of extensions will eventually rewrite their software to be thread safe.

Exactly what is meant by "thread safe" is difficult to explain because there are numerous implications. A couple of issues that relate, however, include the addressing of a process's memory space and the startup code of a DLL.

When an application runs, it is given a chunk of memory with which the process can do what it wants. Anything that exists in the memory space, the process can use. When a DLL is loaded, it is put into the memory of the process that loaded it. At this point, the process accesses the DLL's functions and variables as if they were part of the program. If a function in the DLL starts a new thread of which the calling program is unaware, however, problems can arise. If this thread alters some variable, for example, the calling program might be totally unaware of this and expect the variable to be something else. If the program were "aware" of the possibility that a thread might alter this variable, it would never make assumptions about a variable's state.

Another "thread awareness" issue has to do with when a DLL is loaded. Every time a Win32 DLL is loaded, a function called `DllMain()` is called automatically. When this function is called (by the system), the DLL will know that it is being loaded and should initialize itself. The `DllMain()` function is also called when a new thread is started, when a thread terminates, and when the DLL is unloaded from memory. A thread-safe extension would use this function to know when a thread is started and to mark certain variables as related to the thread. An example of this as a problem was the `Win32::ODBC` extension. Before it was rewritten to be thread safe, problems occurred when running it under `PerlIS.dll` on a web server. `PerlIS.dll` uses a different thread to run each Perl script. If two scripts were running at the same time that were ODBC databases accessed, the `Win32::ODBC` extension would overwrite certain global variables. This caused script errors and even crashes in some cases. Now that the extension is thread safe, each thread has its own set of variables that it, and only it, uses.

## Security

Security is always an issue with CGI scripts, but the most blaring and obvious security risk is to have a physical copy of `perl.exe` in a CGI directory. I cannot emphasize enough how dangerous this is—so don't do it!

If some hacker discovers that you have a Perl executable in your CGI directory (or somewhere else in your web tree), all he has to do is submit some command like this:

```
http://www.foo.com/cgi-bin/perl?-e+"`kill.exe+*`"
```

If your server has `kill.exe` somewhere in its path, your server will crash in moments because the command terminates (or "kills") all system processes. Or maybe the hacker could submit something like this:

```
http://www.foo.com/cgi-bin/perl?-e+"unlink(glob('*'))"
```

This could erase all the files in your `cgi-bin` directory. Of course, depending on how your other security settings are set up, this may or may not be a problem. It is quite clear, however, that placing the Perl executable anywhere in your web tree is just an unnecessary security risk. Even if you have all sorts of security set, chances are good that a hacker could still submit the following:

```
http://www.foo.com/cgi-bin/perl?-
d+"map{open(FILE,$_);while(read(FILE,
$data,2048)){print+$data;}}close(FILE)}glob('*')"
```

This will dump the contents of all files in your `cgi-bin` directory. If you have sensitive data or scripts that would show where data is stored, this indeed could be a security breach.

A safe place to put Perl is anywhere except your web server's web site directories. A common example would be to install Perl into a `c:\perl` directory. Generally speaking, this location is not used as a web directory (at least I hope it isn't). Therefore, only users who have logged on to the machine could execute a Perl script. However, using technologies such as DCOM (refer to [Chapter 5](#)), a user with appropriate permissions could execute Perl (or any application) from a remote machine, regardless of where it is installed.

Another way to help prevent rogue hackers from executing applications (such as `Perl.exe`) is by setting NTFS file and directory permissions so that web site directories do not have Execute permissions. Only Read permissions are required. Refer to [Chapter 11](#) for more details on security and permissions.

Yet another way to prevent hackers from using a web server to execute applications is to configure the web site correctly. This means removing "application" execution settings. Microsoft's IIS has the capability to configure a directory to allow no execution, to run only `.ASP` pages, or to allow program and `.ASP` page execution. If you allow only `.ASP` pages to run, then you are preventing `.exe` files from running.

## ***CGI Script Permissions***

It is important to recognize that each process that runs on a Windows NT machine is granted what is known as a *security token*. This token is like an identification card. When a process tries to do something that is considered to be restricted (such as opening a file, loading a Registry hive, or modifying a user account), the operating system checks the ID card (the security token) of the process to see whether it is allowed to perform the task. If the OS determines that the process does not have the correct credentials (or permissions), the process is not allowed to do what it wants.

Every process has one of these security tokens attached to it, even services. When a user logs on, the system assigns a security token to the user. When the user starts a program, the program is given

a duplicate of the user's security token. This is where processes get these tokens. When a user runs a Perl script from a command line, the Perl process receives a duplicate of the user's security token. This is very important to understand because, when a service starts, the same thing occurs.

When you install a service, you must tell the service when to start (never start, start only when told to do so, or start automatically whenever the system boots up). You can tell the service to start either as a part of the system or as a user (in which case, you supply the user account and password). If configured to start as the system, the service is given a default system security token. Otherwise, it will try to log on as the specified user and will be given a duplicate of the user's security token.

Most web servers run as a service, so they are assigned a security token either from the system or from a specified user account. Suppose a web server runs under an account name of something like `IUSR_TEST`. Assume also that an administrator logs on to the web server and runs a Perl script from a command line. The script runs exactly as expected. The administrator then tries running the same script from a web browser and it fails. The problem most likely is that the administrator has privileges identified in his security token that are needed but that the web server does not have.

The solution to this predicament is to grant the required permissions and privileges to the account under which the web server will run. However, giving a web server's process more permissions than it has by default might be considered a security breach itself.

The things to consider when determining why a script fails when run under a different account (such as by a web server) are file permissions, Registry key permissions, and privileges to administrate accounts and other resources such as network shares, printers, and communication devices.

## Tip

*Sometimes, determining which privileges and permissions are causing problems can be quite difficult because of the many possibilities. One of the quickest ways to help narrow down a cause of contention is to use auditing.*

*From the User Manager program, an administrator can enable system auditing such as logon attempts. The Registry, files, and directories can also be audited. Auditing is enabled by the same dialog box that enables you to change permissions on these same resources.*

*After auditing has been enabled, the Event Viewer's security log will show all the audit tracking information. Be forewarned, however; if auditing is enabled on busy servers, the event log can fill up quite quickly.*

## Network Communications

A problem that occurs when running a Perl script either as a CGI script or as an ASP page (when using PerlScript) is accessing network resources. One would assume that if the web server account has permissions to access a network resource (like a printer, a net share, and so forth), the script could just access the resource. If the web server has read access on the file `\ServerA\Data\products.dat` (assuming that this file is tab-delimited containing product names, prices, and web links), the script in [Example 12.11](#) should work.

### **Example 12.11 Simple CGI script accessing a network resource**

```
01. $ | = 1;
02. my $File = "\\\\ServerA\\Data\\products.dat";
03. print "Content-type: text/html\\n\\n";
04. print "<HTML><HEAD><TITLE>Product List</TITLE></HEAD><BODY>\\n";
05. if( open( FILE, "< $File" ) )
06. {
07.     my @Data = <FILE>;
08.     close( FILE );
09.     print "<UL>\\n";
10.    foreach my $Line ( @Data )
11.    {
12.        chop $Line;
13.        my( $Item, $Price, $Href ) = split( '\\t', $Line );
14.        print "<LI><a href=\"$Href\">$Item</a> ($Price)</LI>\\n";
15.    }
16.    print "</UL>\\n";
17. }
18. else
19. {
20.     print "<H1>Can not open $File: $!</H1>";
21. }
22. print "</BODY></HTML>\\n";
```

Don't be surprised, however, if [Example 12.11](#) doesn't work. For some reason, Microsoft's web servers (IIS version 3.0 and higher) seem to have a problem that, under certain circumstances, fails to open the file (regardless of permission and privilege settings). I have yet to find an adequate answer to explain why this happens. Unless there is a patch to the web server itself that corrects this problem, there does not seem to be a fix. Just be aware that this might be a problem if you choose to write CGI scripts that access files on remote machines. An occasional quick check on Microsoft's web site for a hot fix might be in order if you stumble upon this problem.

## **PerlScript**

PerlScript does not understand the concept of a current directory. Because of this, your CGI scripts must specify full paths, not relative ones. Refer to the section "[PerlIS.dll](#)" (earlier in this chapter) for a more thorough explanation.

## **Win32::NetAdmin**

If an error occurs during a function call, using `GetLastError()` might return an incorrect value if running in a multithreaded environment such as using the `PerlIS.dll` ISAPI application or using PerlScript. The reason for this is that the last error is held in a single variable. If an error occurs, the error number is stored in the variable. It is possible that between the time you call a function and the time you call `GetLastError()`, another script could have been run (a CGI script on a web server, for example) that would change the value of the `LastError` variable. The `Win32::NetAdmin` extension is not thread friendly. For more information about this, refer to the "Threads" section earlier in this chapter.

## Win32::Registry

When you connect to a remote Registry and either save a key to a file or load a key from a file, it is important to note that the path to the file specified is relative to the computer on which the Registry lives. Suppose, for example, that a script connects to `\ServerA` and tries to load a key from `c:\temp\key.dat` using the following command:

```
$Remote_HKEY_LOCAL_MACHINE->Load( "TestKey" , "c:\\temp\\key.dat" );
```

The file that will be loaded is in `\ServerA`'s `c:\temp` directory. If the file exists on the machine running the script but not on `\ServerA`, the function will fail. The same is true for the `Save()` method—the file will be saved on `\ServerA`.

## Win32::ODBC

The `Win32::ODBC` extension requires that 32-bit ODBC (version 2.5 or later) be installed on your machine. This is very important and evidently not so obvious to some users. ODBC may or may not be installed on your machine, depending on which applications are installed. Usually, installing something like Microsoft's Office suite will install a compatible version. If you need to install ODBC, you can obtain it from <http://www.microsoft.com/odbc/>. It is easy to get sidetracked at Microsoft's ODBC web page because it also promotes other services such as ADO, OLEDB, RDO, Universal Database Services, and others. It is important to note that most of these services rely on ODBC and are not replacements for it.

To determine whether you have ODBC installed on your machine, you can go to the Windows Control Panel and look for an ODBC applet. If there is none, you do not have ODBC installed; however, it is possible to have it installed without a Control Panel applet—if you accidentally deleted the applet from your `system32` directory. If this is the case, you'd better reinstall it anyway.

After you have ODBC installed, you need to install ODBC drivers. Generally, you need an ODBC driver for each type of database you will be accessing. If you need access to an Oracle database, for example, you need an Oracle ODBC driver but not the SQL Server driver (unless, of course, you also need to access a SQL Server database). Some drivers can handle multiple types of databases, like the MS Access driver that manages Access, FoxPro, Excel, and text files. These each appear as different ODBC drivers in the Control Panel applet, but they all use the same `.DLL` files.

Several third-party ODBC driver companies produce high-performance drivers that can (usually) speed up queries on a database more than the standard driver that comes with a database. These companies also provide drivers for other platforms such as Macintosh and UNIX.

## Permissions

One common problem that usually appears on web servers is incorrect permissions. When an administrator writes a script using `Win32::ODBC`, he might find that the script works when he tests it but fails when accessing it as a CGI script or when other users access the script. This is typically due to inadequate permissions.

A user who runs a script using `Win32::ODBC` must have (at least) read permissions on the ODBC Manager and ODBC driver files. These are usually held in the `C:\WINNT\SYSTEM32` directory.

Additionally, permissions must be granted on and sometimes in the actual database itself. Sometimes this is not a practical matter, such as when dealing with a database server such as Oracle or SQL Server. These types of databases typically require the database administrator to apply permissions on the database system as opposed to permissions placed on files (as in the case of FoxPro and Paradox databases).

Some ODBC drivers also require read and write permissions on certain directories and files. The Access ODBC driver, for example, needs write access on the temporary directory (usually `C:\TEMP`) so that it can create, access, and delete temporary files. Additionally, it needs write access to the directory in which the database file resides. This is because the driver will create and manage an `.LDB` file that contains information regarding who is currently accessing the database file. This caveat makes it difficult to interact with Access database files on CD-ROMs.

## Win32::OLE

One of the most notable things I have discovered with the `Win32::OLE` extension is the way methods return values (or, I should say, do not return values). This deserves some clarification.

Most all Perl functions return some sort of value to indicate either success or failure. `Win32::OLE` is one of those rare exceptions. Now this really is not the extension's fault because there is no way for the extension to know what the return value would represent. If a COM object's method did not return any value, how could the `Win32::OLE` extension possibly know that the method is supposed to not return any value? For this reason, the extension is playing it smart by not making assumptions.

Suppose that I am playing around with MS Word, and I create a document object. My code wants to save the document, so it calls the following:

```
$Result = $Doc->Save();
```

After the line is executed, the `$Result` variable will be empty regardless of whether the `Save()` method was successful. This is because the `Save()` method does not return any value to my Perl script. So how am I to know whether the save actually worked? By using the `LastError()` method.

I can always query the `Win32::OLE->LastError()` method to discover whether my last interaction with a COM object was successful or whether an error occurred. If the method returns a `0`, there is no error. If it returns some error message, I know there was a problem. Based on this, I should rewrite my code that saves the Word document:

```
01. $Doc->Save();
02. if( $Error = Win32::OLE->LastError() )
03. {
04.   print "An error occurred while saving document: $Doc-
>{Name}\n";
05.   print "$Error\n";
06. }
```

# Summary

With all that Win32 Perl is capable of doing, it is difficult to find reasons to be dissatisfied with it. It would be wonderful to write a script that just works as you expect it to, but we live in an imperfect world in which we have to consider all possibilities, lest we find that our scripts fail when we need them the most.

All sorts of Win32 Perl resources are available on the Internet. Many of these provide tips for troubleshooting as well as the latest details on bug fixes. Some of the more common ones include the following:

- **ActiveState Tool Corp.** (<http://www.activestate.com/>). This is the ActiveState web site. Here you can find most of the updated information regarding Win32 Perl (a.k.a. ActivePerl). This is also the place to go to subscribe to the many Win32 Perl List servers such as the following:
  - Win32-Admin (administrative issues)
  - Win32-Database (database issues)
  - Win32-Users (general usage issues)
  - Win32-web (web issues such as CGI scripts)

You can also search the online database of messages for each group.

- **The Official Perl Home Page** (<http://www.perl.com/>). The official Perl web site. Here is where you can find articles, announcements, links, and other neat-o things, including the source for Perl 5.005.
- **The Comprehensive Perl Archive Network** (<http://www.cpan.org/>). Long heralded as *the* place to find Perl-related modules, extensions, and scripts. This archive is mirrored all over the Internet, and most mirrors are kept well up-to-date. If you are looking for a script or extension that performs a particular task, try looking here first.
- **Joe Casadonte's Perl for Win32 page** (<http://www.northbound-train.com/perlwin32.html>). This is a collection of links, scripts, extensions, tutorials, utilities, programs, and the list goes on. From here you can also hop onto the Perl web ring. This site is a must for anyone from a novice to a full-fledged guru.
- **Jutta Klebe's Perl page** (<http://www.bbyte.de/jmk/>). Jutta's site explains the `Win32::PerlLib` extension with details, examples, and links to other related sites. You can also find her binaries here.
- **Dada's Perl Lab** (<http://dada.perl.it/>). This is Aldo Calpini's Perl page. He has documented most of his extensions rather well at this site. You can also find binaries for these extensions as well as information regarding future versions and new extensions.
- **Roth Consulting's Perl Page** (<http://www.roth.net/perl/>). This site has documentation for most of the company's modules and extensions. Additionally, it has a Perl links page and other interesting information including the `Win32::AdminMisc` and `Win32::ODBC` FAQs.
- **Philippe Leberre's Perl Pages** (<http://www.le-berre.com/>). Here you can find some really good breakdowns of some of the common extensions. This is a good source for information; however, it is generally technical in detail, so beginners might be a bit awestruck.
- **Robin's Perl for Win32 Page** (<http://www.geocities.com/SiliconValley/Park/8312/>). This resource contains links, explanations, and some sample scripts to help get you started with Win32 extension fundamentals.
- **Amine Moulay Ramdane's Perl for Win32 Modules Site** (<http://www.generation.net/~aminer/Perl/>). This site has a wealth of useful modules. These are for the hard-code programmer but can be of use to anyone.

- **The Perl Journal** (<http://www.tpj.com/>). Sure this does not have much online help for Win32 Perl, but it is *the* journal for any Perl hack. It covers all platforms, including Win32. A subscription is fairly cheap, and it is well worth the price for the knowledge that comes in each issue.
- **Perl Month online magazine** (<http://www.perlmonth.com/>). This is an interesting monthly dedicated to the Perl language. It has articles that sometimes include Win32-specific issues.
- **Windows Scripting Solutions Journal** (<http://www.win32scripting.com/>) This is a great print and electronic monthly journal that covers all sorts of different Win32 scripting solutions. It also has a terrific monthly Win32 Perl column.
- **Usenet** (<news:comp.lang.perl.misc>). This is about the most informative group of Perl users, addicts, and hackers that you can find. Sometimes this group can get quite ornery, and flame wars are common; with a little netiquette, however, your problems can quickly find resolutions.
- **Usenet** (<news:comp.lang.perl.modules>). Don't be fooled; this group also covers extensions. Most posts in this group pertain only to modules and extensions. It's a wonderful resource.
- **Google.com** (<http://www.google.com/>). This one might not be so obvious, but you would be surprised at how many other non-Perl-related groups discuss Perl issues (possibly even problems you might have). Google now manages the Usenet database formally run by Deja.com.

## Appendix A. Win32 Perl Resources

There are a number of Perl resources. This appendix lists some favorites.

### Book Resources

This book has several scripts that can take a considerable amount of time to type in by hand. Instead, you might want to just download them from my book's web site:

<http://www.roth.net/books/extensions/>

As with all programming-related projects, this book is bound to have a few bugs. As they become evident, I will update my book errata web page:

<http://www.roth.net/books/extensions/errata/>

There are several other Perl books worth noting. I keep an ongoing list of these books on my recommended reading web page:

<http://www.roth.net/books/>

### Web Resources

The following sections list some favorite web resources.

#### The Official Perl Web Site

<http://www.perl.com/>

This is the official Perl web site. Here you can find news, articles, book reviews, code, modules, pretty much anything related to Perl.

## CPAN

<http://www.cpan.org/>

The Comprehensive Perl Archive Network (CPAN) is the global network of Perl code. Authors post modules, scripts, and extensions here, and they are replicated around the world. Chances are, someone has already written code to do what you need. You would look for it here.

## ActiveState Tool Corporation

<http://www.activestate.com/>

These are the guys you can thank for porting Perl to Win32. There have been many Win32 ports, but this team has been championing it for years. Here you can find Perl binaries, FAQs, email list archives, and other interesting information.

## Microsoft Developer Network

<http://msdn.microsoft.com/>

You might not expect it, but this site has a few articles on Perl. More importantly, it has the entire Win32 API documented online! When you are struggling with the `Win32::API` extension, you can check your syntax and values with this web site.

This site also empowers you with the capability to search all of the Microsoft KnowledgeBase articles. This can quickly solve some headaches for more than just Win32 Perl-related woes.

## Perl for Win32

<http://www.northbound-train.com/perlwin32.html>

This is *the* place for Win32 Perl administrators. If you have not yet visited this site, do so now! Not only that, bookmark it as well. Here you can find everything from the latest modules to book reviews. This site is a perfect jumping point with tons of links to various Perl-related sites.

## dada's perl lab

<http://dada.perl.it/>

This is Aldo Calpini's Perl site. He has produced some of the more influential Win32 extensions such as `Win32::API` and `Win32::GUI`.

## Roth Consulting

<http://www.roth.net/>

This is the home for many Win32 extensions, a book guide, a `Win32::ODBC` FAQ, a script repository, Perl links, and other Perl-related information.

## Amine Moulay Ramdane's Perl for Win32 Modules Site

<http://www.generation.net/~aminer/Perl/>

Amine has contributed a flurry of exciting, hard-core modules.

## Usenet Resources

The Usenet is a smorgasbord of information waiting to be consumed. Here you can read what people have written, or you can post a message yourself.

You might have to check with your Internet service provider to discover which Usenet server to access. If you don't have direct access to the Usenet, you can access it through a Usenet web site such as Google.com (<http://groups.google.com/>).

The following Usenet groups are good sources of Perl discussions:

- **comp.lang.perl**. This is a now-defunct general Perl discussion group. Even though some postings still are made to this group, most serious Perl users frequent the `comp.lang.perl.misc` group.
- **comp.lang.perl.announce**. Here's a good place to read what new modules and extensions are being introduced.
- **comp.lang.perl.misc**. This is a miscellaneous Perl discussion group. It is where most people post messages.
- **comp.lang.perl.modules**. This group is where you can post messages regarding Perl modules and extensions.
- **comp.lang.perl.moderated**. This useful group discusses most any Perl topic. Because it's moderated, don't be surprised if traffic is low and the messages are concise and well thought out.
- **alt.perl**. This is an alternative Perl discussion group. You can find some pretty interesting stuff here.
- **alt.perl.sockets**. This group discusses networking and socket-related Perl issues.

In addition to the Internet Usenet groups, you can read and post messages on the Roth Consulting Usenet server, `news://news.roth.net/`. There are several groups covering Win32-specific issues as well as others.

## List Servers

Another very popular way to communicate with fellow Perl coders is to participate in the various listserv lists. These lists are email-based lists that distribute a user's posted message to the entire list of participants.

## ActiveState Tool Corp

[http://www.activestate.com/Support/Mailing\\_Lists/index.html](http://www.activestate.com/Support/Mailing_Lists/index.html)

ActiveState has its own list server where Win32 Perl folks participate.

## **Topica.com**

<http://www.topica.com/>

Topica is a list server portal site that defines what it calls "channels" (its equivalent for a list). Hop over to this site and run a search on the keyword "perl." There are many Perl-related lists to choose from.

## **Perl Web Ring**

<http://www.perlring.org>

For those who are not familiar with WebRings, they are basically rings of web sites all linked together. Each WebRing shares a common theme; this one has the Perl theme, of course.

# **Electronic Magazines and Journals**

This section lists some popular online magazines and journals.

## **The Perl Journal**

<http://www.itknowledge.com/tpj/>

TPJ is a tried-and-true mainstay that is an absolute must for any administrator.

## **Perl Month**

<http://www.perlmonth.com/>

This is an ongoing journal that has interesting articles and features.

## **Web Techniques Magazine**

<http://www.webtechniques.com/>

Even though this is not a Perl-only magazine, it does cover Perl topics from time to time. Either way, it is interesting reading.

## **Windows Scripting Solutions**

<http://www.winscriptingsolutions.com/>

This sister publication of *Windows 2000* magazine is a great scripting newsletter. It covers a wide variety of scripting languages such as Perl and Visual Basic.

## **The Code Project**

<http://www.codeproject.com/>

A good collection of forums, reviews, code, and other neat stuff for developers.

## Win32 Extensions

Over the past few years, an absolute wealth of Win32-specific Perl extensions have been released into the community. This has been great for system administrators and coders for obvious reasons. However, the problem of lack of decent documentation persists more than ever. With so many great modules available, it quickly became difficult to decide which ones to document and which ones to leave out of this book.

The criteria I used to decide which extensions to focus on came pretty much from what was already documented in the first edition and what is currently being discussed on various online forums. I would love this book to be the place where readers can learn about the latest extensions, but to do that would require several volumes.

This section illustrates the syntax for functions within many of the popular Win32 extensions. Each function's proper syntax is followed by a brief description of what the function does and a description of the return values.

### Win32::AdminMisc

The `Win32::AdminMisc` extension provides miscellaneous administrative functions. This is a hodgepodge of functions that either were missing from other extensions or were modifications of existing functions.

#### ***CreateProcessAsUser( \$CommandString [, \$DefaultDirectory] [, %Config] )***

This function creates a process based on the specified parameters that will be running under the account you are impersonating with `LogonAsUser()`. If successful, the function returns the new process's PID; otherwise, it returns a `FALSE` value.

#### ***DelEnvVar( \$Name [, \$Type [, \$Timeout] ] )***

This function deletes the specified environmental variable of the specified type using the specified timeout (for applications to acknowledge the change). The effect of this function will be seen by all running (non-DOS) programs globally. If successful, the function returns `TRUE`; otherwise, it returns a `FALSE` value.

#### ***DNSCache( [1/0] )***

This function sets the local DNS cache on `(1)` or off `(0)`. If nothing is specified, it returns only the current state of the DNS cache. The function returns the new state of DNS caching (either `1` [caching enabled] or `0` [caching disabled] value).

#### ***DNSCacheCount()***

This function returns the current number of cached elements. This cannot exceed the value of `DNSCacheSize()`. `DNSCacheCount()` returns the number of cached elements in the DNS cache.

#### ***DNSCacheSize( [\$Size] )***

This function sets the local DNS cache size to the specified size. `DNSCacheSize( [$Size] )` returns the new (or current) DNS cache size.

### ***ExitWindows( \$Flags )***

This function either logs off the current user or shuts down Windows. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***GetComputerName()***

This function returns the name of the computer. This performs the same function as the `Win32::GetNodeName()` found in the Win32 module. If successful, this function returns the computer's network name; otherwise, it returns `undef`.

### ***GetDC( [ \$Domain / \$Server ] )***

This function returns a domain controller (primary or backup) for the specified domain. If successful, this function returns a domain controller's network name; otherwise, it returns `undef`.

### ***GetDriveGeometry( \$Drive )***

This function returns an array consisting of drive information for the specified drive. If successful, this function returns an array; otherwise, it returns `undef`.

### ***GetDrives( [ \$DriveType ] )***

This function returns an array of drive roots of the specified type. If successful, this function returns an array; otherwise, it returns `undef`.

### ***GetDriveSpace( \$Drive )***

This function returns an array consisting of the total drive capacity and the available space on the specified drive. If successful, this function returns an array; otherwise, it returns `undef`.

### ***GetDriveType( \$Drive )***

This function returns an integer describing the type of drive specified. If successful, this function returns an integer representing the drive type or a `1` if unable to determine the type; otherwise, it returns a `FALSE` value.

### ***GetEnvVar( \$Name / \%Hash [, \$Type ] )***

This function returns the value of the specified environmental variable based on the specified type. If successful, this function returns the variable's value; otherwise, it returns `undef`.

### ***GetFileInfo( \$File, \%Info )***

This function retrieves extended file information (such as copyright, original name, company name) for the specified file. If successful, this function populates the specified hash with file-related information and returns `TRUE`; otherwise, it returns a `FALSE` value.

## **GetGroups( \$Machine \$GroupType, ( \@List / \%%List ) [, \$Prefix ] )**

This function retrieves the names of all groups that match the specified type and, optionally, that match the specified prefix. If successful, this function populates either a hash or an array and returns `TRUE`; otherwise, it returns a `FALSE` value.

## **GetHostAddress( \$DNSName ), GetHostName( \$IPAddress ), gethostbyname( \$DNSName ), and gethostbyaddr( \$IPAddress )**

These four functions are the same but go by different names for backward compatibility. They return either an IP address or a DNS name, depending on what is passed into the function. If successful, they return an IP address or DNS name; otherwise, they return a `FALSE` value.

## **GetIDInfo()**

This function returns an array containing process and thread ID information. If successful, the function returns an array; otherwise, it returns a `FALSE` value.

## **GetLogonName()**

This function returns the name of the user this account is logged on as. This is not necessarily the same as the account the Perl script is running under. An account can log on as another user (known as "impersonating" another account). If successful, this function returns the current user account name; otherwise, it returns `undef`.

## **GetMachines( \$Machine, \$MachineType, ( \@List / \%%List ) [, \$Prefix ] )**

This function retrieves an array or a hash populated with the names of computers of the specified type. The list will optionally only consist of those machine names that begin with the specified prefix. The list is generated by the specified machine. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **GetMemoryInfo()**

This function retrieves a hash of memory-related information. If successful, this function returns a hash; otherwise, it returns `undef`.

## **GetPDC( [ \$Domain / \$Server ] )**

This function returns the primary domain controller (PDC) of the specified domain. If successful, this function returns the name of the PDC; otherwise, it returns `undef`.

## **GetProcessorInfo()**

This function retrieves a hash of processor-related information. If successful, this function returns a hash; otherwise, it returns `undef`.

## **GetStdHandle( \$Handle )**

This function retrieves the Win32 handle to the specified standard handle specified in handle. If successful, this function returns the Win32 handle; otherwise, it returns `undef`.

### ***GetTOD( \$Machine )***

This function retrieves the time of day from the specified machine. If successful, this function returns the time in Perl time format; otherwise, it returns `undef`.

### ***GetUsers( \$Server, \$Prefix, ( \@List / \%%List ) )***

This function retrieves an array or a hash populated with the names of user accounts that match the specified prefix. If successful, this function returns `true`; otherwise, it returns a `FALSE` value.

### ***GetVolumeInfo( \$Drive )***

This function retrieves a hash of drive volume information for the specified drive. If successful, this function returns a hash; otherwise, it returns `undef`.

### ***GetWinVersion()***

This function retrieves a hash of Windows versions. If successful, this function returns a hash; otherwise, it returns `undef`.

### ***LogoffAsUser( [1/0] )***

This function logs the current account out from an "impersonated" account. The logout can use force if specified, which forces the logoff to occur even if applications are currently open. This function always returns `true`.

### ***LogonAsUser( \$Domain, \$User, \$Password [, \$LogonType ] )***

This function logs the current account on under the specified user account in the specified domain using the specified password. The logon type can be specified as well. If successful, this function returns `true`; otherwise, it returns a `FALSE` value.

### ***ReadINI( \$File [, \$Section [, \$Key]] )***

This function retrieves data from an `INI` file based on the specified file, section, and key. If successful, this function returns an array or a string; otherwise, it returns `undef`.

### ***RenameUser( \$Domain / \$Machine, \$UserName, \$NewUserName )***

This function renames the specified user account on the specified machine. If successful, this function returns `true`; otherwise, it returns a `FALSE` value.

### ***ScheduleAdd( \$Machine, \$Time, \$DOM, \$DOW, \$Flags, \$Command )***

This function schedules the specified command to execute at the specified time on the specified machine. If successful, this function returns the new scheduled job number; otherwise, it returns `undef`.

### ***ScheduleDel( \$Machine, \$JobNumber [, \$MaxJobNumber ] )***

This function removes the specified job number (or numbers) from the specified machine. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***ScheduleGet( \$Machine, \$JobNumber, \%\%JobInfo )***

This function retrieves information about the specified scheduled job on the specified machine. If successful, this function populates the hash and returns `true`; otherwise, it returns a `FALSE` value.

### ***ScheduleList( \$Machine [, \%\%List ] )***

This function retrieves the total number of jobs scheduled to run on the specified machine. Optionally, a hash is populated with information regarding each job. If successful, this function returns the number of jobs listed; otherwise, it returns `undef`.

### ***SetComputerName( \$Name )***

This function sets the network name for the current computer. If successful, this function returns the new computer name; otherwise, it returns a `FALSE` value.

### ***SetEnvVar( \$Name, \$Value [, \$Type [, \$Timeout ] ] )***

This function sets the type-specified environmental variable to have the specified value. Optionally, the variable will be of the specified type and will have a specified timeout value for applications to acknowledge the modification. If successful, this function returns `true`; otherwise, it returns a `FALSE` value.

### ***SetPassword( (\$Machine / \$Domain), \$User, \$Password )***

This function sets the password for the specified user on the specified machine. This can be performed only by an administrator. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***SetVolumeLabel( \$Drive, \$Label )***

This function sets the label on the specified drive. If successful, this function returns `true`; otherwise, it returns a `FALSE` value.

### ***UserChangePassword( \$Domain, \$User, \$OldPassword, \$NewPassword )***

This function modifies the password for the specified user on the specified machine (or domain). If successful, this function returns `true`; otherwise, it returns a `FALSE` value.

### ***UserCheckPassword( (\$Machine / \$Domain), \$User, \$Password )***

This function verifies whether the specified password is indeed the password for the specified user account on the specified machine (or domain). There are many reasons for this function to fail; check the documentation. If successful (password is validated), it returns `true`; otherwise, it returns a `FALSE` value.

### ***UserGetAttributes( (\$Machine / \$Domain), \$UserName, \$UserFullName, \$Password, \$PasswordAge, \$Privilege, \$HomeDir, \$Comment, \$Flags, \$ScriptPath)***

This function retrieves some properties of the specified user account information for the specified user on the specified machine (or domain). If successful, this function returns `true`; otherwise, it returns a `false` value.

### **`UserGetMiscAttributes( $Machine / $Domain, $User, \%Attributes )`**

This function retrieves a hash of account properties for the specified user account on the specified machine (or domain). If successful, this function populates a hash and returns `true`; otherwise, it returns `undef`.

### **`UserSetAttributes( $Machine / $Domain, $UserName, $UserFullName, $Password, $PasswordAge, $Privilege, $HomeDir, $Comment, $Flags, $ScriptPath )`**

This function sets some properties of the specified user account on the specified machine (or domain). If successful, this function returns `true`; otherwise, it returns a `false` value.

### **`UserSetMiscAttributes( $Machine / $Domain, $User, $Attribute1=>$Value1 [, $Attribute2=>$Value2 [, ... ] ] )`**

This function sets attributes of the specified user account on the specified machine (or domain). If successful, this function returns `true`; otherwise, it returns a `false` value.

### **`WriteINI( $File, $Section, $Key, $Value )`**

This function writes data to an `INI` file with the specified file, section, key, and value. If successful, this function returns `true`; otherwise, it returns `undef`.

## **Win32::ChangeNotification and Win32::ChangeNotify**

Both of these extensions are the same; they just have different names. It is suggested that `Win32::ChangeNotify` be used; however, the `Win32::ChangeNotification` exists for backward compatibility.

### **`$Obj->Close()`**

This function closes and terminates a change notification object. This function returns `true`.

### **`$Obj->FindNext()`**

This function clears the current state of the `Win32::ChangeNotification` and advances the change notification queue. This must be called before calling the `Wait()` method. If successful, this function returns `true`; otherwise, it returns `undef`.

### **`$Obj->Wait( $TimeOut )`**

This function waits for a change to take place or until the timeout value expires. The timeout is in milliseconds. If successful, this function returns `false` (0); otherwise, it returns a non `undef` and non 0 value.

### **`FindFirst( $Obj, $PathName, $WathSubTree, $Filter )`**

This function creates a `Win32::ChangeNotification` object looking at the specified path and (possibly) any subtrees. This is all based on the specified filter. If successful, this function sets the object variable (the first parameter) to the newly created object and returns `TRUE`; otherwise, it returns `undef`.

## **Win32::Console**

The `Win32::Console` extension interfaces Win32 Perl with a Win32 console. This enables a script to interact with a DOS-like window.

### **`$Obj->Alloc()`**

This function creates (allocates) a new console for the application. This can be used to display the contents of a console buffer. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` (`undef`) value.

### **`$Obj->Attr( [$Attributes] )`**

This function sets and retrieves the attributes for the current console. If successful, this function returns the new attributes.

### **`$Obj->Clear( [$Attributes] )`**

This function clears the console buffer, optionally filling the buffer with the specified attributes. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` (`undef`) value.

### **`$Obj->Cursor( [$X, $Y [, $Size, $Visible ] ] )`**

This function retrieves and optionally sets the X and Y coordinates of the console buffer's cursor. It optionally sets the size and visibility of the cursor. If successful, this function returns a four-element array of location, size, and visibility of the cursor; otherwise, it returns a `FALSE` (`undef`) value.

### **`$Obj->Display()`**

This function displays the console buffer in the application's console. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` (`undef`) value.

### **`$Obj->FillAttr( $Attribute, $Number, $X, $Y )`**

This function fills the console buffer with the specified number of attributes beginning at the specified location. If successful, this function returns the number of attributes written; otherwise, it returns a `FALSE` (`undef`) value.

### **`$Obj->FillChar( $Character, $Number, $X, $Y )`**

This function fills the console buffer with the specified number of characters beginning at the specified location. If successful, this function returns the number of characters written; otherwise, it returns a `FALSE` (`undef`) value.

### **`$Obj->Flush()`**

This function flushes everything from the console input buffer. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->GenerateCtrlEvent( \$Type, \$ProcessPID )***

This function sends a break signal of the specified type to the specified process. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->GetEvents()***

This function retrieves the number of pending events in the console input buffer. If successful, this function returns the number of pending events; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->Info()***

This function retrieves an array of information related to the console buffer, including its size, position, attributes, and cursor-related information. If successful, this function returns an array; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->Input()***

This function retrieves an array of events from the console input buffer. If successful, this function returns an array of events; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->InputChar( \$Number )***

This function reads the specified number of characters from the console input buffer. If successful, this function returns the read characters; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->InputCP( [\$CodePage] )***

This function retrieves and optionally sets the console's code page. If successful, this function returns the code page; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->MaxWindow( [\$Flag] )***

This function retrieves the maximum X and Y coordinates that the console buffer allows. If successful, this function returns a two-element array; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->Mode( [\$Mode] )***

This function retrieves and optionally sets the input and output modes of the console. If successful, this function returns the new mode; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->MouseButtons()***

This function retrieves the number of buttons on the computer's mouse. If successful, this function returns the number of mouse buttons; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->OutputCP( [\$CodePage] )***

This function retrieves and optionally sets the output buffer code page. If successful, this function returns the new output code page; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->PeekInput()`**

This function retrieves the list of events from the console input buffer without removing the events from the input buffer. If successful, this function returns an array of events; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->ReadAttr( $Number, $X, $Y )`**

This function retrieves the specified number of attributes from the specified location within the console buffer. If successful, this function returns the read attributes; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->ReadChar( $Number, $X, $Y )`**

This function retrieves the specified number of characters from the specified location within the console buffer. If successful, this function returns a string of character data; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->ReadRect( $Left, $Top, $Right, $Bottom )`**

This function reads data (both characters and attributes) from within the specified rectangle. The returned data is used with the `WriteRect()` method. If successful, this function returns a string of data; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->Scroll( $Left, $Top, $Right, $Bottom, $X, $Y, $Character, $Attribute [, $ClipLeft, $ClipTop, $ClipRight, $ClipBottom] )`**

This function moves the data (characters and attributes) from the specified rectangle to the location specified (by `$X` and `$Y`). Any empty space left is filled with the specified character and attribute. Optionally, a clipping region can be specified. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->Select( $StdHandle )`**

This function redirects the specified standard buffer to the console buffer (`$Obj`). If successful, this function returns `TRUE`; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->Size( [$X, $Y] )`**

This function retrieves and optionally sets the console buffer size. If successful, this function returns a two-element array; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->Title( [$Title] )`**

This function changes the title bar's text for the application's console. If successful, this function returns the new (or current) title; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->Window( $Flag, $Left, $Top, $Right, $Bottom )`**

This function retrieves and optionally sets the application's console window size. The specified parameters either are relative to the window's current coordinates or are absolute depending on the specified flag. If successful, this function returns a four-element array; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->Write( $Text )`**

This function writes the specified text string to the console buffer. If successful, this function returns the number of characters written; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->WriteAttr( $Attributes, $X, $Y )`**

This function writes the specified attributes to the specified location in the console buffer. If successful, this function returns the number of attributes written; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->WriteChar( $Text, $X, $Y )`**

This function writes the specified text to the specified location in the console buffer. If successful, this function returns the number of characters written; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->WriteInput( @Events )`**

This function writes the specified list of events to the console input buffer. If successful, this function returns the number of characters written to the input buffer; otherwise, it returns a `FALSE (undef)` value.

### **`$Obj->WriteRect( $Data, $Left, $Top, $Right, $Bottom )`**

This function writes the specified rectangle data to the specified window. The data is obtained by calling the `ReadRect()` method. If successful, this function returns a four-element array (indicating the affected window size); otherwise, it returns a `FALSE (undef)` value.

## **`new Win32::Console(), new Win32::Console( $StdHandle ), new Win32::Console( $AccessMode, $ShareMode )`**

The `new Win32::Console()` function creates a new default console buffer.

The `new Win32::Console( $StdHandle )` function creates a new console buffer object based on the specified standard console buffer.

The `new Win32::Console( $AccessMode, $ShareMode )` function creates a new console buffer in memory with the specified access and share modes. If successful, these functions return a reference to a `Win32::Console` object; otherwise, they will return `undef`.

## **`Win32::EventLog`**

The `Win32::EventLog` extension interacts with the Win32 event log. This works specifically on Windows NT (and also on Windows 95 and 98 if they have the domain admin programs installed).

### **\$Obj->Backup( \$File )**

This function creates a backup copy of the event log by saving it to the specified file. If successful, this function returns `true`; otherwise, it returns a `false` value.

### **\$Obj->Clear( \$File )**

This function creates a backup of the event log to the specified file and then clears the log of all records. If successful, this function returns `true`; otherwise, it returns a `false` value.

### **\$Obj->GetNumber( \$Record )**

This function retrieves the total number of records in the event log. If successful, this function sets the value of the first parameter to the total number of records and returns `true`; otherwise, it returns a `false` value.

### **\$Obj->GetOldest( \$Record )**

This function retrieves the oldest record that exists in the event log. If successful, this function stores the record number in the first parameter and returns `true`; otherwise, it returns a `false` value.

### **\$Obj->Read( \$Flags, \$RecordOffset, \%Event )**

This function reads one entry from the event log based on the specified record offset and flags. If successful, this function returns a hash reference that is stored in the third parameter and returns `true`; otherwise, it returns a `false` value.

### **\$Obj->Report( \%EventRecord )**

This function writes the event defined by the specified event record to the event log. If successful, this function returns `true`; otherwise, it returns a `false` value.

### ***new( \$Source, \$Machine ), Open( \$Source, \$Machine )***

Both functions create a new event log object on the specified machine referencing the specified event log source. If successful, they return an event log object; otherwise, they return a `false` value.

## **Win32::File**

This extension provides general file-attribute management. A series of constants represents the different attributes of a file.

### ***GetAttributes( \$Path, \$Attributes )***

This function retrieves the attributes from the specified file or directory. If successful, this function assigns `$Attributes` with the value representing the different attributes and returns `true` (1); otherwise, it returns a `false` (0) value.

### ***SetAttributes( \$Path, \$Attributes )***

This function sets the attributes for the specified file or directory. If successful, this function sets the attributes on the specified path and returns `true` (1); otherwise, it returns a `false` (0) value.

## Win32::FileSecurity

This extension manages permissions placed on files and directories. This will work only on a Windows NT machine.

### ***EnumerateRights( \$Mask, \@Rights )***

This function translates a permission bitmask into an array of permission strings. If successful, this function populates the array with permission strings and returns `true`; otherwise, it returns a `false` value.

### ***Get( \$Path, \%Permissions )***

This function retrieves the permissions (DACL) from the specified file or directory. If successful, this function populates the hash with permission information and returns `true`; otherwise, it returns a `false` value.

### ***MakeMask( @Permissions )***

This function creates a permission bitmask from a list of permission strings (by using the extension's constants). If successful, this function returns a bitmask; otherwise, it returns a `false` (`undef`) value.

### ***Set( \$Path, \%Permissions )***

This function sets the permissions (DACL) on the specified file or directory. If successful, the path's permissions are set, and the function returns `true`; otherwise, it returns a `false` value.

## Win32::IPC

This extension is a base extension used by `Win32::ChangeNotification` (and `Win32::ChangeNotify`), `Win32::Mutex`, `Win32::Semaphore`, and `Win32::Process`. It has no useful methods or functions that a script calls directly.

The other listed extensions inherit functions from `Win32::IPC`, which they use internally.

## Win32::Lanman

This extension provides an interface into the various network-related functionality that Windows NT/2000/XP exposes. Its name comes from the support for Microsoft's legacy Lan Manager product. Many of Win32's networking-related interfaces are based on the Lan Manager code.

This extension exports more than 230 functions, far too many to list in this appendix. However, they are all documented in the `Lanman.pm` module file. This documentation is in POD format and is not only accurate but very concise and well written with plenty of examples. Please refer to this documentation.

## **Win32::Message**

This extension provides an interface into the Win32 NetBIOS messaging API. It enables you to register network names and send messages.

### **NameAdd( [\$Machine,] \$Name )**

This function adds the specified network name to the specified machine (or local host by default). If successful, this function returns `trUE`; otherwise, it returns a `FALSE (0)` value.

### **NameDel( [\$Machine,] \$Name )**

This function removes the specified network name from the specified machine (or local host by default). If successful, this function returns `TRUE`; otherwise, it returns a `FALSE (0)` value.

### **NameEnum( [\$Machine] )**

This function returns an array of the registered network names for the specified machine (or local host by default). If successful, this function returns an array of network names; otherwise, it returns a `FALSE (0)` value.

### **Send([ \$Machine,] \$Receiver, \$Sender, \$Message )**

This function sends the specified message to the specified receiver from the specified sender. The specified machine will actually send the message (provided the account executing the command has permissions to do so). If the message is successfully sent (this does not necessarily mean it was successfully received), the function returns `trUE`; otherwise, it returns a `FALSE (0)` value.

## **Win32::Mutex**

This extension creates and manages Win32 MUTEX (MUTually EXclusive) objects.

### **Create( \$MutObj, \$InitialOwner, \$Name )**

This function opens a MUTEX object that can be used by any running process. If the MUTEX does not currently exist, it is created.

The value of `$InitialOwner` determines whether the calling thread (because Perl does not truly support threads yet considers this to be the calling process) owns the MUTEX. A nonzero value means the process indeed does own the MUTEX. Typically, this value will be `1`. If successful, this function returns `trUE` and sets `$MutObj` to the new MUTEX; otherwise, it returns a `FALSE` value.

### **Open( \$MutObj,\$Name)**

This function opens an existing MUTEX with the specified name. If successful, this function returns `trUE`, and `$MutObj` is set to the MUTEX object; otherwise, it returns a `FALSE` value.

**`$Obj->Release();`**

This function releases the MUTEX object. If no other process has attached to the MUTEX, it is destroyed. If successful, this function returns `trUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->wait( [$Timeout] )`**

This method will wait until the MUTEX becomes signaled or until the timeout value expires. If successful, this function returns `trUE`; otherwise, it returns a `FALSE` value.

## **Win32::NetAdmin**

The `Win32::NetAdmin` extension provides administrative, network, domain, and account management functions.

### **`GetAnyDomainController( $Server, $Domain, $Name )`**

This function discovers the name of any domain controller (either a PDC or a BDC). If successful, `$Name` will contain the name of the domain's PDC, and the function returns `TRUE`; otherwise, it returns `undef`.

### **`GetDomainController( $Server, $Domain, $Name )`**

This function discovers the name of a primary domain controller (PDC). If successful, `$Name` will contain the name of the domain's PDC, and the function returns `TRUE`; otherwise, it returns `undef`.

### **`GetServers( $Server, $Domain, $Flags, \@ServerList )`**

This function retrieves the list of machine names that satisfy the specified flags. If successful, this function populates the array with computer names and returns `trUE`; otherwise, it returns `undef`.

### **`GetUsers( $Server, $Filter, \@UserList )`**

This function obtains a list of user accounts on the specified server that satisfy the specified filter. If successful, this function returns `TRUE`, and the array is populated with the names of user accounts; otherwise, it returns `undef`.

### **`GroupAddUsers( $Server, $Group, $User )`**

This function adds the specified user to the global group on the specified server. If the third parameter is an array reference, all of the user accounts listed in the array are added to the global group. If successful, this function returns `TRUE`; otherwise, it returns `undef`.

### **`GroupCreate( $Server, $Group, $Comment )`**

This function creates a global group on the specified server giving it the specified comment. If successful, this function returns `trUE`; otherwise, it returns `undef`.

### **`GroupDelete( $Server, $Group )`**

This function deletes a global group from the specified server. If successful (the global group exists and was deleted), it returns `trUE`; otherwise, it returns `undef`.

### ***GroupDeleteUsers( \$Server, \$Group, \$Users)***

This function removes a user account from the specified global group on the specified server. If the third parameter is an array reference, all the user accounts listed in the array are removed from the global group. If successful, this function returns `TRUE`; otherwise, it returns `undef`.

### ***GroupGetAttributes( \$Server, \$Group, \$Comment )***

This function retrieves the comment from the specified global group that exists on the specified server. If successful, this function sets the third comment variable (the parameter) with the global group's comment and returns `TRUE`; otherwise, it returns `undef`.

### ***GroupGetMembers( \$Server, \$Group, \@UserList )***

This function populates the specified array with the list of user accounts that make up the global group. If successful, the array is populated with account names, and the function returns `TRUE`; otherwise, it returns `undef`.

### ***GroupIsMember( \$Server, \$Group, \$User )***

This function checks that the user is a member of the global group on the specified server. If the user exists in the global group, it returns `TRUE`; otherwise, it returns `undef`.

### ***GroupSetAttributes( \$Server, \$Group, \$Comment )***

This function sets the comment for the global group on the specified server. If successful, this function returns `TRUE`; otherwise, it returns `undef`.

### ***LocalGroupAddUsers( \$Server, \$Group, \$User )***

This function adds the specified user to the local group on the specified server. If the third parameter is an array reference, all the user accounts listed in the array are added to the local group. If successful, this function returns `TRUE`; otherwise, it returns `undef`.

### ***LocalGroupCreate( \$Server, \$Group, \$Comment )***

This function creates a local group on the specified server giving it the specified comment. If successful, this function returns `TRUE`; otherwise, it returns `undef`.

### ***LocalGroupDelete( \$Server, \$Group )***

This function deletes a local group from the specified server. If successful, it returns `TRUE`; otherwise, it returns `undef`.

### ***LocalGroupDeleteUsers( \$Server, \$Group, \$Users )***

This function removes a user account from the specified local group on the specified server. If the third parameter is an array reference, all the user accounts listed in the array are removed from the local group. If successful, this function returns `TRUE`; otherwise, it returns `undef`.

### ***LocalGroupGetAttributes( \$Server, \$Group, \$Comment )***

This function retrieves the comment from the specified local group that exists on the specified server. If successful, this function sets the third comment variable (the parameter) with the local group's comment and returns `TRUE`; otherwise, it returns `undef`.

### ***LocalGroupGetMembers( \$Server, \$Group, \@UserList )***

This function populates the specified array with the list of user accounts that make up the local group. If successful, the array is populated with account names, and the function returns `trUE`; otherwise, it returns `undef`.

### ***LocalGroupIsMember( \$Server, \$Group, \$User )***

This function checks that the user is a member of the local group on the specified server. If the user exists in the local group, it returns `trUE`; otherwise, it returns `undef`.

### ***LocalGroupSetAttributes( \$Server, \$Group, \$Comment )***

This function sets the comment for the local group on the specified server. If successful, this function returns `trUE`; otherwise, it returns `undef`.

### ***UserChangePassword( \$Domain, \$Username, \$OldPassword, \$NewPassword )***

This function changes the password of the specified user account on the specified server. This function can be run from any user. If successful, this function returns `TRUE`; otherwise, it returns `undef`.

### ***UserCreate( \$Server, \$UserName, \$Password, \$PasswordAge, \$Privilege, \$HomeDir, \$Comment, \$Flags, \$ScriptPath )***

This function creates a user account on the specified server. If successful, this function returns `TRUE`; otherwise, it returns `undef`.

### ***UserDelete( \$Server, \$User )***

This function deletes a user account from the specified server. If successful, this function returns `TRUE`; otherwise, it returns `undef`.

### ***UserGetAttributes( \$Server, \$UserName, \$Password, \$PasswordAge, \$Privilege, \$HomeDir, \$Comment, \$Flags, \$ScriptPath )***

This function retrieves the comment, home directory, flags, password, password age, privilege, and logon script path for the specified user on the specified server. If successful, this function returns `TRUE` and sets each of the passed-in variables with their respective values; otherwise, it returns `undef`.

### ***UserSetAttributes( \$Server, \$UserName, \$Password, \$PasswordAge, \$Privilege, \$HomeDir, \$Comment, \$Flags, \$ScriptPath )***

This function sets the comment, home directory, flags, password, password age, privilege, and logon script path for the specified user on the specified server. If successful, this function returns `TRUE` and modifies the specified user account; otherwise, it returns `undef`.

## ***UsersExist( \$Server, \$UserName )***

This function checks to see whether the specified user account exists on the specified server. If successful (the account exists), it returns `TRUE`; otherwise, it returns `undef`.

## **Win32::NetResource**

The `Win32::NetResource` extension provides network resource-management functions.

### ***AddConnection( %NetResource, \$Password, \$UserName, \$Connection )***

This function connects to the specified network resource using the specified username and password. The connection is mapped to the specified device. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***CancelConnection( \$LocalDeviceName, \$Connection, \$Force )***

This function terminates a network connection that is mapped to the specified device. This will be either a temporary or permanent termination based on the value of the connection parameter. Force can be used to terminate the connection even if open resources are using it. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***GetError( \$ErrorCode )***

This function retrieves the error code for previous calls to any `Win32::NetResource` function. If successful, this function sets `$ErrorCode` with the last generated error and returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***GetSharedResources( \@Resources, \$Type )***

This function generates a list of hashes representing all the network resources of the specified type. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***GetUNCName( \$UNC, \$Path )***

This function retrieves the UNC of the shared network resource connected to the specified device. If successful, this function sets the `$UNC` variable with the UNC name and returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***NetShareAdd( %Share, \$ParamError [, \$Machine] )***

This function shares the device specified by the `%Share` hash on the specified machine (local host is the default). The second parameter needs to be a scalar variable but should be ignored. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***NetShareCheck( \$Device, \$Type [, \$Machine] )***

This function checks whether a particular device is shared on the specified machine (local host is the default). If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***NetShareDel( \$ShareName [, \$Machine] )***

This function stops sharing the specified shared device on the specified machine (local host is the default). If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***NetShareGetInfo( \$ShareName, \%Share [, \$Machine] )***

This function retrieves share information regarding the specified shared device residing on the specified machine (local host is the default). If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***NetShareSetInfo( \$ShareName, \%Share , \$ParamError [, \$Machine] )***

This function sets attributes of the specified shared resource on the specified machine (local host is the default). The third parameter needs to be a scalar variable but should be ignored. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***WNetGetLastError( \$ErrorCode, \$Description, \$Name )***

This function retrieves the extended network error specified. This is only applicable for certain types of errors generated by certain functions. If successful, this function sets `$Description` and `$Name` with the error description and error name, respectively, and returns `TRUE`; otherwise, it returns a `FALSE` value.

## **Win32::ODBC**

This extension provides an interface into the ODBC API.

### ***\$Obj->Catalog( \$Qualifier, \$Owner, \$Name, \$Type)***

This function creates a data set that contains table information about the DSN based on the specified criteria. If successful, this function generates a data set and returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->Close( [\$Result] )***

This function closes the ODBC connection. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->ColAttributes( \$Attribute [, @FieldNames ] )***

This function returns the specified attribute on each of the specified fields in the current record set. If successful, this function returns the attribute; otherwise, it returns `undef`.

### ***\$Obj->ConfigDSN( \$Option, \$Driver, \$Attribute1 [, \$Attribute2 [, \$Attribute3 ...]] )***

This function configures a DSN with the specified parameters. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->Connection()***

This function returns the connection number associated with the ODBC connection. If successful, this function returns `true`; otherwise, it returns a `false` value.

### **`$Obj->Data( [ $FieldName [, $FieldName2 ... ] ] )`**

This function returns the contents of columns with names `$FieldName`, `$FieldName2`, and so forth, or the current row (if nothing is specified). This method is available only for backward compatibility. The preferred method is `DataHash()`. If successful, this function returns an array of column values; otherwise, it returns `undef`.

### **`$Obj->DataHash( [ $Field1, $Field2, $Field3, ... ] )`**

This function returns the contents for specified fields in a hash. If no fields are specified, all fields are returned. If successful, this function returns a hash of field names and values; otherwise, it returns `undef`.

### **`$Obj->DataSources( [$DSN] )`**

This function returns a hash of data sources and ODBC remarks about them. If successful, this function returns a hash of data sources and remarks; otherwise, it returns a `false` value.

### **`$Obj->Drivers()`**

This function returns a hash of ODBC drivers and their attributes. If successful, this function returns a hash of drivers and attributes; otherwise, it returns a `false` value.

### **`$Obj->DropCursor( [ $CloseType ] )`**

This function drops the cursor associated with the ODBC object, forcing the cursor to be deallocated. If successful, the function returns `true`; otherwise, it returns a `false` value.

### **`$Obj->Error(), Error()`**

The `Error()` function returns the last-encountered error. The returned value is context dependent.

It returns a string or an array, depending on the return context containing error information.

### **`$Obj->FetchRow( [$Row [, $Type] ] )`**

This function retrieves the next record from the current dataset, optionally specifying the number of rows to be fetched (default is 1) and the type of fetch (default is a relative fetch). If successful, this function returns `true`; otherwise, it returns a `false` value.

### **`$Obj->FieldNames()`**

This function retrieves all the field names in the current dataset. If successful, this function returns an array of field names; otherwise, it returns `undef`.

### **`$Obj->GetConnections()`**

This function returns an array of connection IDs showing which connections are currently open. If successful, this function returns an array of connection IDs; otherwise, it returns `undef`.

### ***\$Obj->GetConnectOption( \$Option )***

This function retrieves the value of the specified connection option. If successful, this function returns the option value; otherwise, it returns `undef`.

### ***\$Obj->GetCursorName()***

This function retrieves the name of the current cursor. If successful, this function returns the cursor's name; otherwise, it returns `undef`.

### ***\$Obj->GetDSN( [\$DSN] )***

This function returns the configuration for the specified DSN. If no DSN is specified, the current connection is used. If successful, this function returns a hash of DSN configuration data; otherwise, it returns `undef`.

### ***\$Obj->GetFunctions( [\$Function1, \$Function2, ... ] )***

This function returns a hash of values indicating the ODBC driver's capability to support specified functions. If no functions are specified, a 100-element associative array is returned containing all possible functions and their values. If successful, this function returns a hash; otherwise, it returns `undef`.

### ***\$Obj->GetInfo( \$Option )***

This function retrieves the value of the specified option. If successful, this function returns the option value; otherwise, it returns `undef`.

### ***\$Obj->GetMaxBufSize()***

This function retrieves the current allocated limit for the `MaxBufSize`. If successful, the function returns the current buffer size.

### ***\$Obj->GetSQLState()***

This function retrieves the SQL state as reported by the ODBC manager and driver. If successful, the function returns the SQL state.

### ***\$Obj->GetStmtCloseType( [\$Connection] )***

This function returns the type of closure that will be used every time the statement is freed. If successful, the function returns the closure type.

### ***\$Obj->GetStmtOption( \$Option )***

This function retrieves the value of the specified statement option value. If successful, the function returns the statement option value.

## **\$Obj->MoreResults()**

This function reports whether there is data yet to be retrieved from the query. If data remains, the function returns `trUE`; otherwise, it returns a `FALSE` value.

## **`new( $ODBCObject / $DSN ) [, ($Option1, $Value1), ($Option2, $Value2) ...] )`**

This function creates a new ODBC connection based on the specified DSN or the specified ODBC object. All specified connection options will be set to the specified values before the physical connection occurs. If successful, the function returns a `Win32::ODBC` object; otherwise, it returns `undef`.

## **`$Obj->RowCount( $Connection )`**

For `UPDATE`, `INSERT`, and `DELETE` statements, the returned value from this function is the number of rows affected by the request. If successful, the function returns the number of affected rows; otherwise, it returns a `-1` value.

## **`$Obj->Run( $Sql )`**

This function executes the `$Sql` command and dumps to the screen info about it. *This is used primarily for debugging.* If successful, the function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **`$Obj->SetConnectOption( $Option, $Value )`**

This function sets the value of the specified connect option to the specified value. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **`$Obj->SetCursorName( $Name )`**

This function sets the name of the current cursor. If successful, this function returns `trUE`; otherwise, it returns a `FALSE` value.

## **`$Obj->SetMaxBufSize( $Size )`**

This function sets the `MaxBufSize` for a particular connection. If successful, this function returns the new buffer size.

## **`$Obj->SetPos( $Row [, $Option, $Lock] )`**

This function moves the cursor to the specified row within the current keyset (not the current data/result set). If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **`$Obj->SetStmtCloseType( $Type [, $Connection] )`**

This function sets a specified connection's statement close type for the connection. If successful, this function returns the new value; otherwise, it returns `undef`.

## **`$Obj->SetStmtOption( $Option, $Value )`**

This function sets the value of the specified statement option. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->Sql( $SQLString )`**

This function executes the specified SQL command. If successful, this function returns `FALSE`; otherwise, it returns a `TRUE` value.

### **`$Obj->TableList( $Qualifier, $Owner, $Name, $Type )`**

This function returns the catalog of tables available to the DSN based on the specified criteria. If successful, this function generates a data set and returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->Transact( $Type )`**

This function forces the ODBC connection to perform a rollback or commit transaction. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **Win32::Perms**

The `Win32::Perms` extension provides functionality to modify Win32 permissions.

### **`$Obj->Add( \%Hash [, \%Hash2 [, ...] ] / $Account, $Mask, $Type, $Flag )`**

This function adds permissions to a `Win32::Perms` object. It returns the number of permissions (ACE entries) added.

### **`$Obj->AddAllow( $Account, $Mask, $Flag )`**

This function adds permissions to a `Win32::Perms` object that grant access to a user. It returns the number of permissions (ACE entries) added.

### **`$Obj->AddAudit( $Account, $Mask, $Flag )`**

This function adds audit information to a `Win32::Perms` object. It returns the number of permissions (ACE entries) added.

### **`$Obj->AddDeny( $Account, $Mask, $Flag )`**

This function adds permissions to a `Win32::Perms` object that denies access to a user. It returns the number of permissions (ACE entries) added.

### **`$Obj->Close()`**

This function closes the `Win32::Perms` object.

### **`DecodeFlag( ( $Flag / \%Hash ) [, \@Flags ] )`**

This function decodes a particular flag value into an array of flag constant names. It returns the total number of flag constants returned.

### ***DecodeMask( ( \$Mask / \%Hash ) [, \@Permissions, \@FriendlyNames ] )***

This function decodes a particular permissions mask into arrays of permission constant names and friendly names. It returns the total number of permission constants returned.

### ***DecodeType( ( \$Type / \%Hash ) [, \@Types ] )***

This function decodes a particular type value into an array of type constant names. It returns the total number of type constants returned.

### ***\$Obj->Dump( [ \@Array ] )***

This function retrieves an array of hash references containing permission configuration data (ACE entries). If nothing is passed in, it prints a dump of the permission settings to the screen; this is used for debugging. Use the `Get()` method instead of this method. It returns the total number of entries returned.

### ***\$Obj->Get( [ \@Array ] )***

This function retrieves an array of hash references containing permission configuration data (ACE entries). It returns the total number of entries returned.

### ***\$Obj->GetDacl( ACL\_RELATIVE / ACL\_ABSOLUTE )***

This function exports the security descriptor's DACL. It returns a binary DACL.

### ***\$Obj->GetSacl( ACL\_RELATIVE / ACL\_ABSOLUTE )***

This function exports the security descriptor's SACL. It returns a binary SACL.

### ***\$Obj->GetSD( [ ACL\_RELATIVE / ACL\_ABSOLUTE ] )***

This function exports the security descriptor. It returns a binary SD.

### ***\$Obj->GetType()***

This function returns the type of Win32 object that the `Win32::Perms` object represents. The value returned is numeric and equates to one of these: `PERM_TYPE_NULL`, `PERM_TYPE_NONE`, `PERM_TYPE_UNKNOWN`, `PERM_TYPE_FILE`, `PERM_TYPE_REGISTRY`, `PERM_TYPE_SHARE`, `PERM_TYPE_PRINTER`, or `PERM_TYPE_SD`.

### ***\$Obj->Group( [\$Account] )***

This function retrieves or sets the group of a security descriptor (`Win32::Perms` object). It returns the (new) owner account.

### ***IsContainer()***

This function checks to see if the `Win32::Perms` object is a container object (such as a directory or Registry key). If it is a container, this method returns `TRUE`; otherwise, it returns a FALSE value.

### ***IsValidAcl( \$BinaryAcl )***

This function checks to see if the binary ACL is valid and well defined. If the ACL is valid, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***IsValidSD( \$SecurityDescriptor )***

This function checks to see if the passed-in SD is valid and well defined. If the SD is valid, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->Import( \$File / \$Object )***

This function imports the security information from a file or `Win32::Perms` object. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->ImportDacl( \$BinaryDACL )***

This function imports the security information from a binary DACL. It returns the number of permission entries (ACE entries) imported.

### ***\$Obj->ImportSacl( \$BinarySACL )***

This function imports the auditing information from a binary SACL. It returns the number of audit entries (ACE entries) imported.

### ***LookupDC( [ 1 / 0 ] )***

This function queries or sets the flag indicating that each user looked up must talk with a domain controller. This function returns the current state of the `LookupDC` flag.

### ***new( [ \$Path / \$Object [, \$Type ] ] )***

This function creates a new `Win32::Perms` object. If the function is successful, a `Win32::Perms` object is returned; otherwise, it returns `undef`.

### ***\$Obj->Owner( [ \$Account ] )***

This function retrieves or sets the owner of a security descriptor (`Win32::Perms` object). It returns the (new) owner account.

### ***\$Obj->Path( [ \$Path ] )***

This function retrieves or sets the path of the `Win32::Perms` object. It returns the current path.

### ***\$Obj->Remove( -1 / \$User / \$Index [, \$User2 / \$Index2 ...] )***

This function removes permissions (ACE entries) from the `Win32::Perms` object. It returns the number of permissions (ACE entries) added.

### ***ResolveAccount( \$TextSid / \$BinarySid [, \$Machine ] )***

This function resolves a SID into a user account. It returns the user account text string.

### ***\$ResolveSid( \$Account [, \$BinarySid [, \$Machine ] ] )***

This function resolves a user account into a text-based SID. Optionally, it can set a scalar variable to the binary SID. It returns the SID in text format.

### ***\$Obj->Set( [ \$Path [, \$Recurse [, \$Container ] ] ] )***

This function commits the permissions represented by the `Win32::Perms` object to the specified object. It returns the number of permissions (ACE entries) that were committed.

### ***\$Obj->SetDacl( [ \$Path [, \$Recurse [, \$Container ] ] )***

This function commits only the DACL (permission information) represented by the `Win32::Perms` object to the specified object. It returns the number of permissions (ACE entries) that were committed.

### ***\$Obj->SetGroup( [ \$Path [, \$Recurse [, \$Container ] ] )***

This function commits only the group account represented by the `Win32::Perms` object to the specified object. It returns the number of groups that were committed.

### ***\$Obj->SetOwner( [ \$Path [, \$Recurse [, \$Container ] ] )***

This function commits only the owner account represented by the `Win32::Perms` object to the specified object. It returns the number of owners that were committed.

### ***\$Obj->SetRecurse( [ \$Path [, \$Container ] ] )***

This function commits the permissions represented by the `Win32::Perms` object to the specified object. This is the equivalent of the `Set()` method, except that it automatically recoursees into sub container objects. It returns the number of permissions (ACE entries) that were committed.

### ***\$Obj->SetSacl( [ \$Path [, \$Recurse [, \$Container ] ] )***

This function commits only the SACL (auditing information) represented by the `Win32::Perms` object to the specified object. It returns the number of auditing ACE entries that were committed.

## **Win32::Pipe**

The `Win32::Pipe` extension provides Win32-specific named pipe management.

### ***\$Obj->BufferSize()***

This function retrieves the current buffer size of the named pipe. If successful, this function returns the size of the buffer; otherwise, it returns a `FALSE` value.

### ***CallNamedPipe( \$PipeName, \$SendData, \$RecieveData [, \$TimeOut ] )***

This function connects as a client to the specified pipe, sends the specified data, and then waits for a response. If successful, this function sets the third parameter with the data read from the pipe and returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->Connect()`**

This function waits for a client to connect to the pipe. If successful, this function returns a `TRUE` value; otherwise, it returns a `FALSE` value.

### **`$Obj->Close( [$NoDisconnect] )`**

This function closes and terminates the named pipe. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->Connect()`**

This function waits for a client to connect to the named pipe. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->Disconnect( [$iPurge] )`**

This function disconnects from the named pipe. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->Error( [$PipeHandle] )`**

This function retrieves the last error generated by the specified instance of the named pipe. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->GetInfo()`**

This function retrieves various bits of information regarding the named pipe, such as the number of instances and the username of the client. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`new Win32::Pipe( $Name [, $Timeout [, $State ] ] )`**

This function creates a named pipe with the specified name, timeout value, and state. If successful, this function returns a new `Win32::Pipe` object; otherwise, it returns a `FALSE` value.

### **`$Obj->Peek( )`**

This function retrieves up to the specified number of bytes from the named pipe without removing the data from the pipe. If successful, this function returns up to the specified number of bytes of data; otherwise, it returns a `FALSE` value.

### **`PeekPipe( $PipeHandle, $Size )`**

This function retrieves up to the specified number of bytes from the specified named pipe without removing the data from the pipe. If successful, this function returns an array consisting of up to the

specified number of bytes of data, the number of bytes returned, and the amount of data waiting that has not been peaked; otherwise, it returns a `FALSE` value.

### **`$Obj->Read()`**

This function reads data from the named pipe. If successful, this function returns the data read from the pipe; otherwise, it returns a `FALSE` (`undef`) value.

### **`$Obj->ResizeBuffer( $Size )`**

This function sets the buffer size of the named pipe to the specified size. If successful, this function returns the new size of the buffer; otherwise, it returns a `FALSE` value.

### **`$Obj->Transact( $SendData, $ReadData )`**

This function writes the specified data to the named pipe and then reads data from the pipe. If successful, this function sets the second parameter with the data read from the pipe and returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`TransactNamedPipe( $Handle, $SendBuffer, $ReceiveBuffer )`**

This function writes the specified data to the named pipe and then reads data from the pipe. If successful, this function sets the second parameter with the data read from the pipe and returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->Write( $Data )`**

This function writes the specified data to the named pipe. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **Win32::Process**

The `Win32::Process` extension creates and manages Win32 processes.

### **`Create( $Obj, $AppName, $CommandLine, $Inherit, $CreateFlags, $InitialDir )`**

This function starts a new process using the specified parameters. If successful, this function sets `$Obj` to point to a `Win32::Process` object and returns the new process's PID; otherwise, it returns a `FALSE` value.

### **`$Obj->GetExitCode( $ExitCode )`**

This function retrieves the exit code for the process. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->GetPriorityClass( $Class )`**

This function retrieves the priority class for the process. If successful, this function sets the first parameter's value to the priority class and returns `TRUE`; otherwise, it returns a `FALSE` value.

### **\$Obj->Kill( \$ExitCode );**

This function terminates the process and sets the process's exit code to the specified value. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **\$Obj->Resume()**

This function resumes the process from a suspended state. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **\$Obj->SetPriorityClass( \$Class )**

This function sets the process's priority class to the specified value. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **\$Obj->Suspend()**

This function suspends the process. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **\$Obj->Wait( \$Timeout )**

This function waits for the process to terminate or until the timeout has elapsed. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **Win32::RasAdmin**

The `Win32::RasAdmin` extension provides management functions for remote access services (RAS).

### **ClearStats( \$Machine, \$Port)**

This function clears the statistics on the specified port on the specified machine. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **Disconnect( \$Machine, \$Port )**

This function disconnects the connection on the specified port on the specified machine. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **GetAccountServer( \$Server / \$Domain )**

This function retrieves the name of the RAS account server of the specified domain (or for the specified server). If successful, this function returns the RAS account server name; otherwise, it returns a `FALSE` value.

### **GetErrorString ( \$ErrorNum )**

This function retrieves the error message for the specified error number. This is for RAS-specific errors. If successful, this function returns the error text; otherwise, it returns a `FALSE` value.

### ***GetPorts( \$Server [, %Ports] )***

This function retrieves the number of ports available on the specified server and optionally populates the hash with port information. If successful, this function returns the number of ports; otherwise, it returns a `FALSE` value.

### ***PortGetInfo( \$Server / \$Domain), \$Port, %Hash )***

This function retrieves a hash containing information about the specified server's (or domain's RAS server) ports. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***ServerGetInfo( \$Server, %Info )***

This function retrieves a hash populated with information about the specified server. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***UserGetInfo( \$Machine / \$Domain), \$User, %Info )***

This function populates a hash regarding the specified user from the specified machine (or domain). If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***UserSetInfo( \$Server / \$Domain), \$User, \$Flag1=>\$Value1 [, \$Flag2=>\$Value2 ...] )***

This function sets the specified flags to the specified values for the specified user in the specified server (or domain). If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **Win32::Registry**

The `Win32::Registry` module provides an interface to manage the Win32 Registry.

### ***\$Obj->Close()***

This function closes the `Registry` key. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->Create( \$SubKey, \$Obj )***

This function creates the specified subkey. If it already exists, it is opened. If successful, this function sets `$Obj` to point to a new Registry object and returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->DeleteKey( \$SubKey )***

This function deletes the specified subkey and any values and subkeys it might contain. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->DeleteValue( \$ValueName )***

This function removes the specified value and its associated data from the key. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **\$Obj->GetKeys( !@Array )**

This function retrieves an array of the names of the specified key's subkeys. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **\$Obj->GetValues( !%Hash )**

This function retrieves a hash consisting of keys representing the subkey names. The hash values are anonymous arrays consisting of the value's name, the value's data type, and the value's data (in order). If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **\$Obj->Load( \$Key, \$FileName )**

This function loads the hive from the specified file (on a local hard drive) into root under the specified key name. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **\$Obj->Open( \$SubKey, \$Obj )**

This function opens the specified Registry subkey. If successful, this function sets `$Obj` to point to a new Registry object and returns `TRUE`; otherwise, it returns a `FALSE` value.

## **\$Obj->QueryKey( \$Class, \$NumOfSubKeys, \$NumOfValues )**

This function retrieves and sets the respective values for the key's class (default value), number of subkeys, and number of values. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **\$Obj->QueryValue( \$SubKey, \$Data )**

This function retrieves the default data value for the specified subkey. If successful, this function returns the retrieved data; otherwise, it returns a `FALSE` value.

## **(\$Obj->QueryValueEx( \$ValueName, 0, \$DataType, \$Data )**

This function retrieves the data and data type for the specified value from the key. Both the `$DataType` and `$Data` variables will be set with their respective values. The second parameter is reserved and must be set to `0`. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **\$Obj->Save( \$File )**

This function saves the key and all its subkeys and values to a file on a local hard drive. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **\$Obj->SetValue( \$SubKey, \$Type, \$Data )**

This function sets the specified subkey's default value and data type. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **\$Obj->SetValueEx( \$Name, \$Reserved, \$Type, \$Data )**

This function sets the specified named value for the key to the specified data and data type. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **Win32::Semaphore**

The `Win32::Semaphore` extension creates and manages Win32 semaphore objects.

### **`Create( $Obj, $Initial, $Max, $Name )`**

This function creates a semaphore object. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->Open( $SemObject, $Name )`**

This function opens an already-created named semaphore. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->Release( $Count, $LastValue )`**

This function releases ownership of a semaphore object. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`$Obj->wait( $TimeOut )`**

This function waits for ownership of a semaphore object. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **Win32::Service**

The `Win32::Service` extension provides basic management functions for Windows services.

### **`GetServices( $Machine, \@List )`**

This function populates an array with the names of the services on the specified machine. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`GetStatus( $Machine, $ServiceName, $Status )`**

This function retrieves the current status of the specified service on the specified machine. If successful, this function sets the value of the `$Status` variable and returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`PauseService( $Machine, $ServiceName )`**

This function pauses the specified service on the specified machine. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### **`ResumeService( $Machine, $ServiceName )`**

This function resumes the paused service on the specified machine. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$StartService( \$Machine, \$ServiceName )***

This function starts the specified service on the specified machine. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$StopService( \$Machine, \$ServiceName )***

This function stops the specified service on the specified machine. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

## **Win32::Shortcut**

The `Win32::Shortcut` extension interfaces the Explorer shell's capability to make and manage shortcuts.

### ***\$Obj->Close()***

This function properly closes a shortcut object. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->Load( \$ShortcutFile )***

This function loads the existing specified shortcut file into the shortcut object. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE (undef)` value.

### ***new Win32::Shortcut( [\$ShortcutFile] )***

This function creates a new explorer shortcut object, optionally based on the existing specified shortcut file. If successful, this function returns a `Win32::Shortcut` object; otherwise, it returns a `FALSE (undef)` value.

### ***\$Obj->Resolve( [\$Flag] )***

This function attempts to resolve the shortcut object, optionally prompting the user for needed details if a flag is specified. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->Save( \$ShortcutFile )***

This function saves the shortcut object to the specified shortcut file. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE` value.

### ***\$Obj->Set( \$Path, \$Arguments, \$WorkingDirectory, \$Description, \$ShowCommand, \$HotKey, \$IconPath, \$IconNumber )***

This function sets the specified attributes on the shortcut object. If successful, this function returns `TRUE`; otherwise, it returns a `FALSE (undef)` value.

## **Win32::WinError**

The `Win32::WinError` extension exports various Windows error constants. The list of exported constants is too numerous to enumerate here. Refer to the `WINERROR.PM` file for this list.